

## INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

# 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



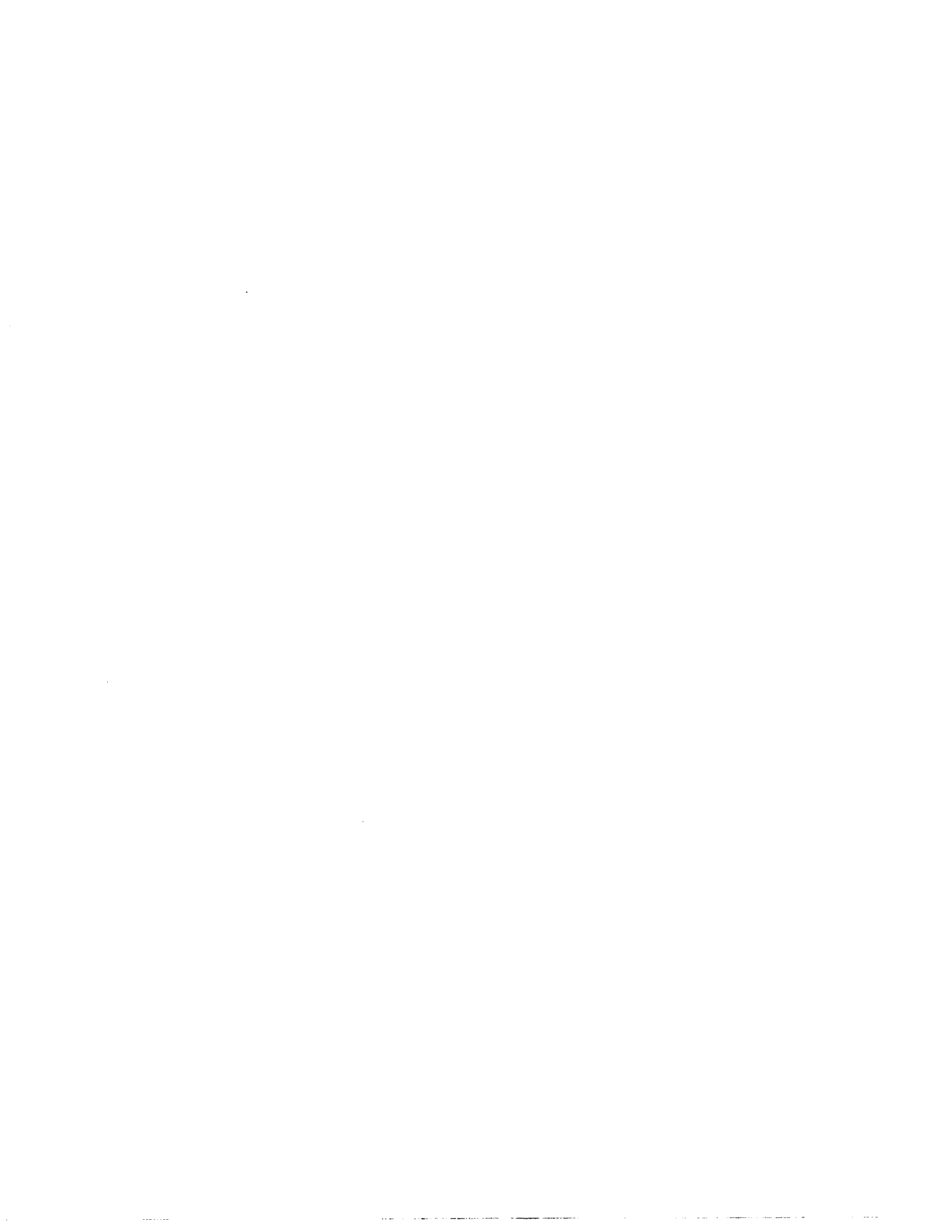
**Order Number 9132521**

**A structural engineering software development environment**

**Zhang, Hong, Ph.D.**

**Purdue University, 1991**

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



**PURDUE UNIVERSITY**  
**GRADUATE SCHOOL**  
**Thesis Acceptance**

This is to certify that the thesis prepared

By Hong Zhang

Entitled

A STRUCTURAL ENGINEERING SOFTWARE DEVELOPMENT ENVIRONMENT

Complies with University regulations and meets the standards of the Graduate School for originality and quality

For the degree of Doctor of Philosophy

Signed by the final examining committee:

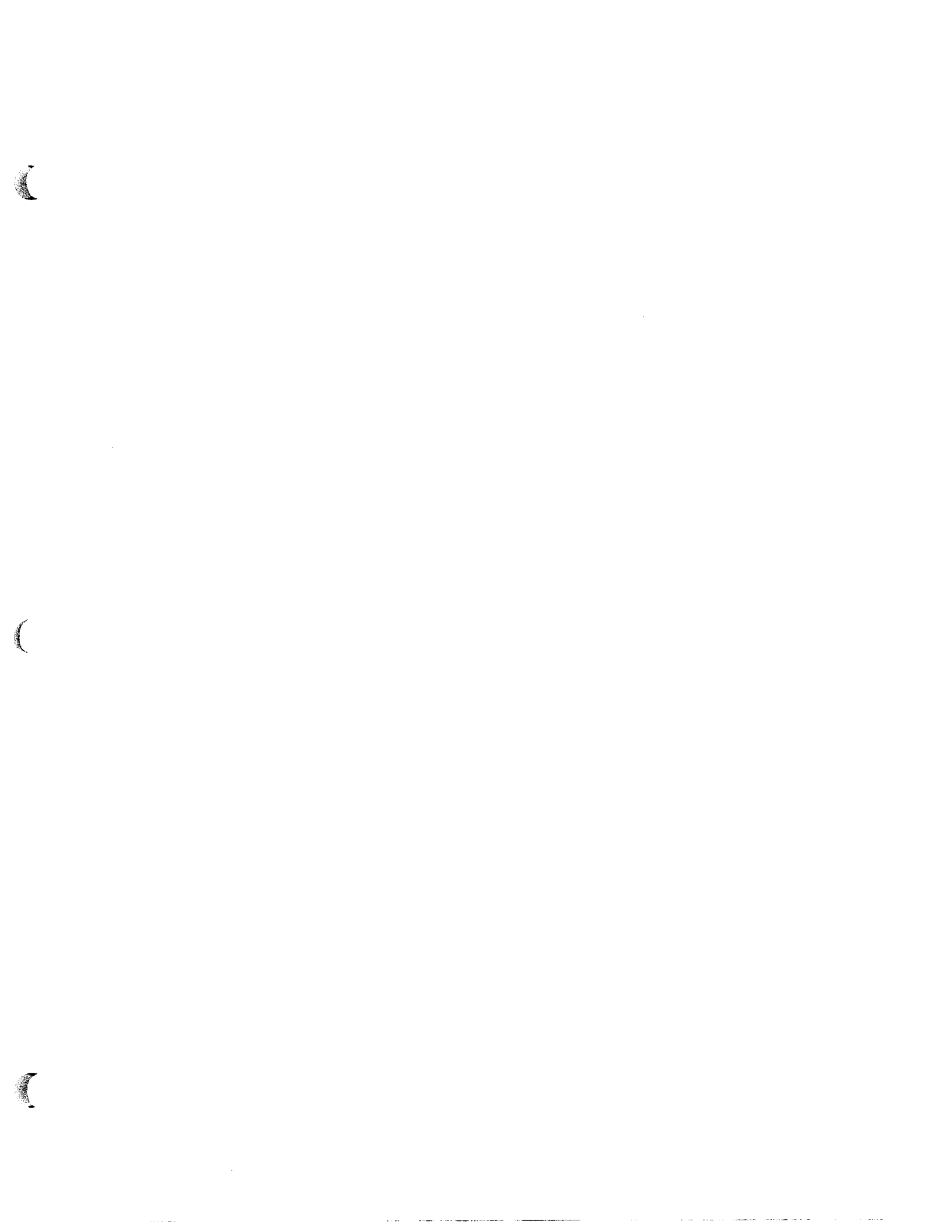
W. J. Chen, chair  
Donald W. White  
Al G. ...  
Robert W. Lee  
G. T. Sun

Approved by:

Harold A. Michael April 29, 1991  
Department Head Date

This thesis  is  is not to be regarded as confidential

W. J. Chen 4/24/91  
Major Professor



A STRUCTURAL ENGINEERING  
SOFTWARE DEVELOPMENT ENVIRONMENT

A Thesis

Submitted to the Faculty

of

Purdue University

by

Hong Zhang

In Partial Fulfillment of the  
Requirements for the Degree

of

Doctor of Philosophy

May 1991

## ACKNOWLEDGEMENTS

The author would like to express his sincere gratitude to his major professors, W.F. Chen and D.W. White, for their support, best guidance, encouragement, advice and assistance during the course of this research.

The author would like to express his sincere gratitude to Professor H.E. Dunsmore for his guidance, advice, and assistance in the aspect of software engineering during the course of this research.

The author is thankful to Professors C.T. Sun and R.H. Lee for their helpful comments and suggestions during the course of this research.

The author expresses a deep appreciation to his wife and son for their endless endurance and care.



## TABLE OF CONTENTS

	Page
LIST OF TABLES.....	xi
LIST OF FIGURES .....	xii
ABSTRACT .....	xvi
CHAPTER 1 INTRODUCTION.....	1
1.1 A Software Crisis in Structural Engineering Computing.....	3
1.1.1 Instructional Software.....	4
1.1.2 Research Software.....	5
1.2 Software Reuse and Domain-Specific Environments.....	7
1.3 A Structural Engineering Software Development Environment.....	10
1.3.1 Motivation .....	10
1.3.2 Development Methodology .....	10
1.3.3 Design Philosophy .....	11
1.3.4 Features .....	11
1.3.5 Benefits .....	12
1.4 Components of the SESDE.....	12
1.4.1 Classification.....	12
1.4.2 A Graphical User-Interface Development System.....	13
1.4.3 A Database Management System .....	15
1.4.4 An Artificial Intelligence System.....	16
1.4.5 Object Classes for Engineering Computing.....	17
1.4.6 Object Classes for Structural Engineering Computing.....	18
1.5 Domain-Specific CASE Tools.....	20
1.5.1 A Tool for Graphical User-Interfaces .....	20
1.5.2 A Tool for Reusable Component Libraries.....	21
1.5.3 A Tool for Application Development .....	21

	Page
1.6 Objective and Scope .....	22
1.7 Organization of the Thesis.....	23

## PART ONE

OVERVIEW OF SOFTWARE ENGINEERING TECHNOLOGIES .....	24
CHAPTER 2 BACKGROUND AND CURRENT ISSUES.....	25
2.1 The Software Life Cycle .....	26
2.2 Characteristics of Well-Engineered Software.....	28
2.3 Current Issues.....	29
CHAPTER 3 SOFTWARE REUSABILITY .....	32
3.1 The Concept of Reuse .....	32
3.2 Characteristics of Reusable Components .....	34
3.3 Design of Reusable Components.....	35
3.3.1 Functional Abstraction .....	36
3.3.2 Data Abstraction.....	37
3.4 Software Reuse and Life Cycle Models .....	39
3.5 Reuse of Components .....	40
3.6 Support of Reuse .....	41
CHAPTER 4 OBJECT-ORIENTED PROGRAMMING.....	43
4.1 Characteristics.....	43
4.1.1 Abstraction .....	44
4.1.2 Encapsulation .....	46
4.1.3 Polymorphism.....	47
4.1.4 Inheritance.....	48
4.1.5 Composition.....	50
4.2 Programming in the C Language.....	51
4.2.1 Representation of Object Classes .....	53
4.2.2 Implementation of Message Passing .....	56
4.2.3 CLOOP: A General Utility for OOP in C .....	62
4.3 Programming in the C++ Language .....	71
4.3.1 Support of Abstraction and Encapsulation .....	71

	Page
4.3.2 Support of Inheritance and Polymorphism .....	73
4.3.3 Support of Composition .....	74
4.4 OOP for Engineering Software Development.....	74
<b>CHAPTER 5 SPECIFIC ISSUES RELATED TO THE SESDE DEVELOPMENT .....</b>	<b>77</b>
5.1 Design of Reusable Components.....	77
5.1.1 Object-Oriented versus Functional Design .....	77
5.1.2 Inheritance versus Composition .....	81
5.1.3 Subsystems.....	83
5.2 Applications Development .....	85
<b>LIST OF REFERENCES.....</b>	<b>86</b>

## PART TWO

<b>A GRAPHICAL USER-INTERFACE DEVELOPMENT SYSTEM.....</b>	<b>89</b>
<b>CHAPTER 6 GRAPHICAL USER INTERFACE TOOLS.....</b>	<b>90</b>
6.1 Introduction .....	91
6.2 The Need for GUI Tools.....	92
6.3 Current State of GUI Development Systems.....	94
6.3.1 User-Interface Toolkits.....	95
6.3.2 User-Interface Development Systems.....	95
6.4 Issues in Design of GUI Tools .....	99
6.4.1 Semantics of the Interface Component.....	100
6.4.2 Communication and Control.....	103
6.5 Case Study: Macintosh Tools.....	105
6.5.1 Macintosh Toolbox.....	106
6.5.2 MacApp Framework.....	108
6.5.3 HyperCard and HyperTalk.....	110
6.6 Case Study: The X11 Toolkit .....	113
6.6.1 Overview .....	113
6.6.2 Widgets.....	114

	Page
6.6.3 Widget Semantics.....	115
6.6.4 Event Handling and Callback Mechanism.....	116
6.6.5 Critique.....	116
6.7 Case Study: GRAFIC/CE88.....	118
6.8 OSF/Motif.....	119
6.8.1 Overview.....	119
6.8.2 The User-Interface Language.....	120
6.8.3 The Motif Resource Manager.....	125
<b>CHAPTER 7 DESIGN ISSUES IN THE DEVELOPMENT OF GUIDES.....</b>	<b>128</b>
7.1 Justification for the research.....	128
7.2 Basic Requirements.....	130
7.3 Design Decisions.....	131
7.3.1 Black-Box versus White-Box Framework.....	131
7.3.2 Windowing-System versus Graphics-System Basis.....	132
7.3.3 Design Methodology.....	133
7.3.4 Internal versus External Control.....	134
7.3.5 Agent Semantics.....	135
7.3.6 Language Binding.....	136
7.4 The HOOPS Graphics Library.....	137
<b>CHAPTER 8 DESCRIPTION OF GUIDES.....</b>	<b>141</b>
8.1 Architecture of the System.....	141
8.2 Callback Manager.....	143
8.3 Event Manager.....	144
8.3.1 Processing of Raw Events.....	145
8.3.2 Handling of Basic GUIDES Events.....	147
8.3.3 The Event Register.....	148
8.3.4 Grabbing of Events.....	148
8.3.5 Queueing of Events from a File.....	149
8.4 GUIDES Agents.....	149
8.4.1 Agent Classes and Agent Groups.....	150
8.4.2 Agent Composition.....	153
8.4.3 States of Agent Instances.....	154
8.4.4 Agent Attributes.....	156
8.4.5 Agent Semantics.....	158
8.4.6 Description of Each Agent Class.....	161

	Page
8.5 Graphical Utilities.....	167
8.6 The Description Language .....	168
8.6.1 Lexical Conventions .....	169
8.6.2 Statements and Compound Statements.....	170
8.6.3 Defining an Agent Instance .....	170
8.6.4 Defining a Composite Agent Instance.....	171
8.6.5 Defining a Restricted Agent Instance .....	172
8.6.6 Connecting the Agent Semantics.....	172
8.6.7 Defining an Agent Style .....	173
8.6.8 Other Features .....	173
8.6.9 A Complete Example .....	174
8.7 GUIDES versus Motif .....	179
LIST OF REFERENCES .....	181

### PART THREE

OBJECT CLASSES FOR ENGINEERING COMPUTING.....	183
CHAPTER 9 THE SESDE OBJECT LIBRARY.....	184
9.1 Object-Oriented Engineering Software Development.....	184
9.1.1 Literature Review.....	184
9.1.2 Reusability .....	187
9.1.3 Efficiency.....	188
9.2 Object Classes in the SESDE Library .....	189
CHAPTER 10 BASIC DATA STRUCTURES AND UTILITIES .....	192
10.1 An Exception Handling Class.....	192
10.1.1 Handling Error Exceptions .....	193
10.1.2 Handling Warning Exceptions.....	196
10.2 Parameterized Array Classes.....	197
10.2.1 The <i>ExtArray(T)</i> Class.....	199
10.2.2 The <i>Bag(T)</i> Class .....	203
10.3 A Parameterized Vector Class.....	207

CHAPTER 11 OBJECT CLASSES FOR FULL MATRICES .....	211
11.1 Introduction .....	211
11.2 Procedural Libraries for Full Matrices .....	213
11.2.1 Abstraction: the Representation of a Matrix .....	213
11.2.2 Dynamic Creation and Destruction of a Matrix.....	215
11.2.3 Utilization of Matrix Characteristics .....	216
11.2.4 Ease of Use and Expressiveness .....	218
11.3 Overview of Full Matrix Classes .....	220
11.3.1 Classification .....	220
11.3.2 Matrix Manipulation Functions.....	221
11.3.3 Use of Two Interfaces for Matrix Operations .....	222
11.3.4 Extensibility .....	224
11.4 Design of the <i>Matrix</i> Class .....	225
11.4.1 Matrix Representation.....	225
11.4.2 Matrix Operations .....	228
11.5 Design of Derived Classes.....	230
11.5.1 The <i>SMatrix</i> Class.....	231
11.5.2 The <i>LUMatrix</i> Class .....	231
11.6 Benchmark Testing and the Matrix Calculator.....	234
CHAPTER 12 OBJECT CLASSES FOR SPARSE MATRICES .....	241
12.1 Introduction.....	241
12.2 Sparse Matrix Manipulation in Procedural Languages.....	242
12.3 Overview of Sparse Matrix Classes.....	247
12.4 Design of the <i>Sparse</i> Class .....	248
12.4.1 Representation of Sparse Matrices.....	248
12.4.2 States of <i>Sparse</i> Objects .....	248
12.4.3 Properties of the <i>Sparse</i> Class .....	251
12.4.4 Classes Used with the <i>Sparse</i> Class.....	251
12.4.5 Interface of the Sparse Classes .....	252
12.5 Design of the <i>ActiveColumn</i> Class.....	254
12.6 Design of the <i>SGraph</i> Class .....	255
12.6.1 The Graph-Based Sparse Storage Scheme .....	256
12.6.2 Determining New Nodal Numbering .....	257
12.6.3 Formulation of Matrix Decomposition.....	259

	Page
12.6.4 Implementation .....	259
12.7 Testing of Sparse Matrix Classes.....	263
LIST OF REFERENCES .....	268
CHAPTER 13 THE SESDE DATABASE MANAGEMENT SYSTEM.....	270
13.1 The Need for Database Management Systems.....	270
13.2 Basic Requirements .....	275
13.3 Older Database Management Technologies .....	277
13.3.1 File Management Systems .....	277
13.3.2 Hierarchical and Network Database Management Systems .....	278
13.3.3 Relational Database Management Systems.....	279
13.3.4 Problems with Older Generations of DBMSs.....	281
13.4 Object-Oriented Database Management Systems .....	283
13.4.1 The Motivation .....	283
13.4.2 Distinguishing Features .....	284
13.4.3 Implementation Approaches.....	287
13.4.4 Limitations of Current ODBMSs.....	290
13.5 Engineering Data .....	293
13.6 Overview of DBMSs for Engineering Software.....	294
13.7 Integrating an ODBMS with the SESDE .....	296
13.7.1 A View Transformation Manager .....	297
13.7.2 An Input Manager.....	297
13.7.3 In Core Object Management .....	299
LIST OF REFERENCES .....	300
CHAPTER 14 SUMMARY, CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK ON SESDE .....	302
14.1 The Problems and the Solution .....	302
14.2 Summary of the Present Work on SESDE.....	305
14.2.1 The Graphical User Interface Development System.....	305
14.2.2 The Generic Object Class Library .....	306
14.2.3 The Object-oriented Database Management System .....	307

	Page
14.3 Specific Recommendations for Follow-up Work on SESDE .....	308
14.3.1 Long-term Tasks .....	308
14.3.2 Short-term Tasks .....	310
VITA.....	312



## LIST OF TABLES

Table	Page
8.1 Callback lists and agent mode changes.....	159
9.1 Object classes developed in the present work .....	190
11.1 Interface of the <i>Matrix</i> class.....	229
11.2 Interface of the <i>SMatrix</i> class.....	232
11.3 Efficiency of $S = D * B$ (Case 1, 2000 operations) .....	238
11.4 Efficiency of $S = B^T D B$ (Case 2, 1000 Operations) .....	238
12.1 Comparison of efficiencies of <i>Sparse</i> classes .....	267

## LIST OF FIGURES

Figure	Page
4.1 Linowes' approach for property inheritance (from Linowes, 1988).....	55
4.2 GUIDES' approach for property inheritance.....	56
4.3 Message-passing function <i>Send</i> (from Linowes, 1988).....	58
4.4 The definition of a self-sufficient object class (from Meyer, 1988).....	59
4.5 A Message passing function of GUIDES.....	61
4.6 Data structure of a general object class in CLOOP.....	63
4.7 Memory organization of an object of a derived class, class C.....	64
4.8 General form of a method.....	66
4.9 Method-dispatch table for a derived class, Class C.....	67
4.10 Class-dispatch table for an application.....	68
4.11 An abstracted representation of <i>SendMessageTo</i> .....	69
5.1 Functional design of the <i>Root</i> component.....	79
5.2 Object-oriented design of the <i>Root</i> component.....	80
6.1 A simplified view of an application that has a GUI.....	100
6.2 Event handling with Macintosh Toolbox.....	107
6.3 A Macintosh resource file.....	109
6.4 A HyperTalk script.....	111
6.5 Dialogue boxes invoked by the (a) <i>ask</i> and (b) <i>answer</i> commands of HyperTalk.....	112

Figure	Page
6.6 Class hierarchy of the widget set distributed by the Project Athena .....	115
6.7 An implementation of the <i>Goodbye world</i> application with the X11 Toolkit.....	117
6.8 The <i>Goodbye world</i> application .....	118
6.9 A sample Motif UIL module.....	122
6.10 Setting up a user-interface specified with UIL .....	126
7.1 Hierarchical structure of HOOPS segments for an application which displays a car in two views .....	138
7.2 Raw events identified by HOOPS.....	139
8.1 The conceptual model of GUIDES .....	142
8.2 The form of GUIDES callback functions.....	143
8.3 Basic events of GUIDES.....	146
8.4 Agent class inheritance hierarchy of GUIDES .....	151
8.5 Semantics of the <i>Button</i> agent .....	160
8.6 A modified version of the <i>Goodbye world</i> application .....	174
8.7 The C code of the <i>Goodbye world</i> application .....	175
8.8 The description file of the <i>Goodbye world</i> application .....	176
10.1 The declaration of the <i>ErrorHandler</i> class .....	194
10.2 Use of the <i>ErrorHandler</i> in the <i>Matrix</i> class.....	195
10.3 A simplified version of the <i>ExtArray(T)</i> class declaration.....	200
10.4 Declaration and implementation of the <i>StringExtArray</i> class .....	201
10.5 Constructing loops over array elements.....	204

Figure	Page
10.6 A simplified version of the <i>Bag(T)</i> class declaration .....	205
10.7 Declaration and implementation of the <i>StringBag</i> class.....	205
10.8 A simplified version of the <i>BagIterator(T)</i> class declaration .....	207
10.9 Constructing loops using <i>BagIterator</i> objects.....	208
10.10 A simplified version of the <i>Vector(T)</i> class declaration .....	209
11.1 A simplified version of the <i>MATRIX</i> class declaration (from Lee, 1989)..	219
11.2 A simplified version of the <i>Matrix</i> class declaration .....	226
11.3 A simplified version of the <i>LUMatrix</i> class declaration .....	233
11.4 An illustration program of the <i>LUMatrix</i> class .....	235
11.5 The C++ benchmark testing programs for Case 2 .....	237
11.6 A script shows the use of the matrix calculator <i>Mac</i> .....	240
12.1 Specification of a set subroutines implementing the skyline scheme .....	243
12.2 An object-oriented design of a sparse matrix component in C.....	245
12.3 A simplified version of the <i>Sparse</i> class declaration .....	249
12.4 A simplified version of the <i>ActiveColumn</i> class declaration .....	255
12.5 A simplified version of the <i>NodeTable</i> class declaration.....	262
12.6 A simplified version of the <i>SGraph</i> class declaration.....	263
12.7 The mesh of the testing case.....	264
12.8 An excerpt of the testing program for <i>Sparse</i> classes.....	265
13.1 Basic relationship: an application with a database .....	271
13.2 Typical configuration of an integrated system.....	272

Figure	Page
13.3 Configuration of an integrated system with local databases .....	296
13.4 Configuration of an integrated system with local databases and a view transformation manager.....	298
14.1 Architecture of the SESDE.....	304

## ABSTRACT

Zhang, Hong. Ph. D., Purdue University, May 1991. A Structural Engineering Software Development Environment. Major Professors: W.F. Chen, D.W. White.

An evolution of the traditional disciplines of structural engineering and computational mechanics driven by the rapid advances in computer technology is currently underway. Research and instruction in these areas are becoming more software dependent and more software intensive. The success and pace of this evolution depends on the rapid and economic development of domain specific applications software.

The SESDE (A Structural Engineering Software Development Environment) is an attempt to provide a systematic support for the development of structural engineering software systems. SESDE is centered around the concept of software reuse, based on object-oriented programming technologies, and composed of reusable software components and domain-specific CASE tools facilitating reuse. The present work focuses on the reusable components, and attempts to build the basic SESDE framework and to establish a model of such an environment which may be useful to other engineering areas.

The reusable components are classified in four groups: (1) a graphical user interface development system (GUIDES); (2) an object-oriented database management system (ODBMS); (3) a generic object class library for engineering computing in general; (4) a structural engineering specific object class library. GUIDES is developed and has been used in research software development and instruction. GUIDES has features which have not been well addressed by existing commercial systems. A set of classes in the generic object class library

is developed. These include classes for general data structures and utilities, for full matrices, and for sparse matrices. Techniques for engineering database management are reviewed. It is concluded that a commercial ODBMS should be integrated and adapted to support the features of the environment. Specific issues associated with the integration are given. Necessary follow-up work of the SESDE are outlined including both long-term development and short-term application of the SESDE components. The long-term tasks are to complete the SESDE system development, which includes the enhancement of the GUIDES, the integration of an ODBMS, the development and enhancement of the structural engineering specific and generic class libraries, and the development of CASE tools. The short-term tasks are focussed on the promotion of the use of existing reusable components.

## CHAPTER 1 INTRODUCTION

Due to rapid advances in the power and potential uses of computers in recent years, traditional engineering disciplines have been undergoing tremendous changes. The rapid advances in workstation technology, characterized by multitasking, networking, large memory and addressing, high-resolution graphics, and interactive graphical user-interfaces, have introduced a new style of computing for engineering research and instruction. This new style of computing offers many advantages over the mainframe and personal computing. Some of these advantages are:

1. User-computer interaction can be accomplished by graphical means, and complex information can be represented with real-time two- and three-dimensional graphics;
2. Multitasking can be accomplished without degradation of performance. The large screen of a workstation can be occupied by one or more windows, and several tasks can be performed simultaneously in these windows; and
3. Collaborative work can be facilitated. A workstation is viewed not as an isolated island but an integral part of a network. Resources such as programs and databases may be shared over the network.

Under the influence of rapidly advancing computer technology, engineering research and instruction are becoming more software dependent and more software intensive. Two challenges are now facing the university computing environment:



- The development, maintenance, and extension of advanced instructional software that will stimulate student interest and learning in an optimum way.
- The full utilization of advanced hardware, the timely development of new methods and approaches in engineering computation, and the use of these tools to provide new insights in engineering research.

The development of high-quality engineering research and instructional software is not keeping pace with the increasing demand for such software, and there is a widening gap between potential and actual computing capabilities. Many existing engineering research and instructional software systems lack extensibility and flexibility for modification. It is therefore difficult to update these systems to take full advantage of new research and teaching developments, and new hardware and/or software capabilities. Moreover, in spite of the availability of improved programming tools, the development of new applications software becomes increasingly difficult due to the added complexity of new applications. A software crisis is apparent in engineering computing in general, and engineering research and education in particular. Specifically, this crisis involves the high cost of software development and maintenance due to inadequate software design. This is especially critical in research because of the dynamic nature of the research environment.

Problems with software development and maintenance became well recognized in the computer science profession in the early 1960's. The discipline of Software Engineering emerged in the late 1960's as a result of the attempts to overcome these problems. Since then, many advancements have been made. Recent progress in the software engineering area includes the development of object-oriented programming methodologies and software development environments with extensive Computer-Aided Software Engineering (CASE) tools. These activities have provided great potential for increasing software

production efficiency and quality in general. Specifically, they have given rise to an unprecedented opportunity to infuse computer technology into all areas of research and instruction, and to give momentum to advancements in engineering science.

However, this potential has not been fully utilized in engineering computing. Much of engineering software still remains on older computer technology and is being outpaced by new software and hardware advances. Also, many of the present software development environments are general purpose in nature. In order to achieve their full potential, these environments need to be combined with specific application-domain tools. Research is urgently needed in the application of advanced software development methodologies to improve software quality and productivity in engineering. This research intends to merge software engineering principles and methodologies to the development of structural engineering software systems.

This chapter gives an overview of the present research. The software related problems and difficulties (i.e., the software crisis) in structural engineering computing are first highlighted in Section 1.1. In Section 1.2, software reuse and software development environments are briefly discussed as a potential solution to the crisis. Section 1.3 outlines an envisioned programming environment for structural engineering software development. Sections 1.4 and 1.5 describe the reusable software components and CASE tools in the envisioned environment. The objective and scope of the present research and the organization of this thesis are described in Sections 1.6 and 1.7 respectively.

### 1.1 A Software Crisis in Structural Engineering Computing

Software systems used in structural engineering computing in the university environment can be generally grouped in two major categories: research systems and Computer-Aided Instructional (CAI) systems. A software

crisis in structural engineering exists that involves the high cost of development and maintenance of these systems due to inadequate software design and development.

#### 1.1.1 Instructional Software

Computer-Aided Instruction (CAI) has become feasible only recently due to the growth of workstation and advanced personal computer technology. The capabilities provided by workstations and "high-end" personal computers such as high-resolution color and gray-scale graphics and ergonomically designed graphical user-interfaces, are necessary features of CAI software. The combination of artificial intelligence techniques with these capabilities (to develop Intelligent Computer-Aided Instructional or ICAI software) is an area of great promise which at present is still in its infancy.

Generally, CAI and ICAI software is difficult to develop, modify, and maintain. CAI and ICAI programs should ideally be developed in a university environment because teaching experience is essential to achieve the desired functionality. However, even with the advances in present software technology, CAI software systems development based on new computer technology still requires a long development period. Due to inadequate software development environments and tight schedules for design, development, and testing, these systems often are low in quality and portability, and they are hard to maintain and modify. Often, there is not a common code base for programs in the same area, and different programs contain a great deal of duplicate coding. Code which accomplishes the same functions is re-developed again and again in new programs. Also, many programs often do not have the flexibility to adapt to evolving computer environments and computer hardware. Therefore, they can easily become obsolete.

### 1.1.2 Research Software

University researchers should take the lead in the investigation and demonstration of approaches which take full advantage of improved computing capabilities. However, to demonstrate new approaches in a timely fashion, university researchers must increase their software productivity.

At the early stage of engineering computation (1960's and 1970's), many software systems were developed by using ad-hoc software development techniques. As engineering software systems increased in size and complexity, problems with ad-hoc approaches in software development, specifically error-proneness and high cost, became apparent. Great efforts have been made to improve software quality and to reduce development and maintenance costs. Various techniques have been proposed including top-down structural programming, database management, problem-oriented languages and virtual machine, documentation quality standards, and use of subroutine libraries. However, the software crisis remains. This is true particularly in the university environment.

At the present time, many existing engineering software systems are one-of-a-kind software. Such systems are built by components which are designed and developed only for a specific application. The design philosophy of many programs is to include all important tools for a particular field in a single system. For example, in a finite element system, many different types of elements and many constitutive models may be included. However, only a few modules in a system are actually used for solving a particular problem. Often, these systems are large, and, if the software is not designed properly, they may contain many complicated interlinked modules. They often may have undocumented dependencies on hardware, operating system, graphics libraries, etc. This makes the maintenance of these systems very difficult.

The extension of these programs to accommodate new techniques or procedures is even more difficult. Because the modules in the system are often closely interlinked, bugs may be introduced in other modules when a module is modified or added to the system. In order to modify or extend one module, one has to understand most, if not all of the modules in the system. However, if one wants to utilize a general purpose system in research, modification is usually unavoidable.

Code duplication is common within and across many of these types of systems. The same piece of code may be developed again and again among different software components and for different software systems because of difficulties in reusing existing software. In a university environment, it is not uncommon that researchers might devote an inordinate amount of their time to software development and maintenance -- time which might be otherwise spent on more fundamental research issues if advanced software engineering approaches and tools were available to facilitate reuse. Thus, the critical issue is lack of software reusability due to improper design and development. Due to the lack of software reusability, efforts spent on software development often cannot be accumulated. This is a great waste of resources and a severe hindrance to advancements in engineering research.

In summary, advances in computer technology provide engineers the potential to speed up their pace and widen their range of engineering research, and to make engineering education more interesting, challenging, and effective. Advances in engineering science have created a demand for high-quality software systems utilizing new hardware capabilities effectively and efficiently. However, due to the difficulties frequently encountered in maintaining existing software and in developing new software, the tremendous increases in computing power offered by modern computers cannot always be fully utilized to meet the demand. At the present time, software development and maintenance are the

main barriers in the infusion of advanced computer technology in engineering research and education. A software crisis involving the excessive time associated with software development and maintenance is thus apparent.

To overcome this crisis and to meet the increasing demands on research and instructional software, research is urgently needed in the application of software engineering principles and methodologies to engineering computing in general and structural engineering in particular. An integrated, domain-specific software development environment centered around the concept of software reuse may help to solve this crisis.

## 1.2 Software Reuse and Domain-Specific Environments

A potential solution to the software crisis is to improve the reusability of software components. Software reuse plays a crucial role in software development because it enables the knowledge about a problem domain to be accumulated and shared. It promises substantial improvement on several aspects: software productivity, maintainability, portability, quality, and standardization.

The idea of software reuse is not new. Creating subroutine libraries such as IMSL is the classical approach for software reusability. However, as will be discussed in Chapter 3, this approach is not sufficient to achieve a large-scale improvement in software quality, productivity, and maintenance. It is also not feasible to decompose existing software systems into reusable components. Reusable components should be carefully designed. Special design and implementation techniques are necessary to achieve reusability. Object-oriented programming appears most promising for attaining reusability of software components. This technique will be discussed further in Chapter 4.

Software development environments, generally known as Computer-Aided Software Engineering (CASE) systems, are a compatible set of tools, usually based on a specific software development methodology. These tools can be employed for several phases of software development and operation. There are hundreds of such systems available in the market, and more become available each year. Currently, most software development environments are general purpose in that they can be applied to any application domain. However, to make CASE systems more useful and attractive to software developers in different application domains, it is necessary to tailor the environments to specific application domains. That is, it is necessary to develop domain-specific software development environments (Dunsmore, 1990).

To take full advantage of previous applications software development, a domain-specific software development environment must support software reuse in the targeted domain. It should consist of a large collection of reusable software components and tools supporting software reuse for the targeted domain. Some knowledge of techniques and practices in the domain of interest has to be embedded in the environment's tools. These features distinguish domain-specific environments from general-purpose environments. Research on domain-specific software development environments is of great interest to both computer scientists and software developers in different application domains.

A domain-specific software development environment for structural engineering computing is the goal of the present research. This environment is named the SESDE, which is an acronym for Structural Engineering Software Development Environment. It is hoped the envisioned environment will make a significant impact on the software quality and productivity in software development for structural engineering research and instruction. The envisioned environment provides general tools that are directly applicable to other areas, and its specific tools should indicate directions for development of domain-specific tools in other areas.

The development of the SESDE is justified due to several important practical reasons as described below.

1. Research is needed to apply software engineering principles and technologies to specific engineering domains and to build domain-specific software development environments. The present work on SESDE demonstrates a possible engineering domain-specific software development environment and its potential in improving software productivity and quality. It provides directions to alleviate many of the current software problems in engineering and scientific research and instruction.
2. The increasing demands on research and instructional software require the development of the SESDE. Although some software development tools are starting to become available in the software market (e.g., standardized graphical user-interface tools, database management systems, and Computer-Aided Software Engineering (CASE) tools), the software tools necessary for the structural engineering software development are currently either inadequate, not portable, or unavailable. The integrated software development environment envisioned will not be commercially available in the foreseeable future.
3. The domain-specific nature requires the development. Reusable software components for structural engineering computing constitute probably the most important part of the environment. These components will be used as basic building blocks for research and CAI software systems. Such a library of reusable software components has not been seen on the commercial market, and it must be developed by structural engineering researchers, staff, and students themselves.



### 1.3 A Structural Engineering Software Development Environment

#### 1.3.1 Motivation

Meyer stated (1988), "Reusability, as a dream, is not new." The author also has had this dream for a long time. The present research is an attempt to make the dream a reality.

To provide systematic support to software development, reusable software components for structural engineering computing have to be identified, designed, implemented, and maintained. A software development environment is necessary which consists of these components and the necessary programming tools, and which provides systematic support for structural engineering software development. These considerations have motivated the development of the domain-specific Structural Engineering Software Development Environment (SESDE).

The envisioned SESDE consists of reusable software components and CASE tools which support software reuse. In the SESDE, the software components will include both structural engineering specific and general-purpose components whose use is not limited to structural engineering. The CASE tools of the SESDE are utilized for managing the software components and helping programmers to find and integrate components into applications.

#### 1.3.2 Development Methodology

Object-Oriented Programming (OOP) is the key methodology employed to achieve the goal of the SESDE. This is because software reusability is the central objective of OOP. This programming methodology facilitates a new style of software development based on large number of prefabricated software components. This new style of software development should be more productive than previous styles. Software developed in this way should be less error-prone, more abstract, more readily modified, and more extendible.

### 1.3.3 Design Philosophy

Where possible, the environment should be built from currently available software components and CASE tools. New software is to be developed where current software is inadequate, incompatible, or unavailable. The research on this environment focuses on the integration of tools that support the development of application-specific programs for both research and instructional activities.

As mentioned previously, object-oriented programming is the major methodology for the software components developed in the environment. The C++ language has been chosen as the major implementation language. However, the environment does not force applications to follow the object-oriented methodology or to use the C++ language. Where possible, interfaces for conventional languages such as C and FORTRAN are provided for software components of the environment.

### 1.3.4 Features

It is important to emphasize that the SESDE itself does not involve the development of general purpose systems for structural analysis, design, etc. Rather, it is an integrated environment for systematic support and development of specific and/or general purpose application programs for both research and instructional activities. The envisioned Structural Engineering Software Development Environment has the following three important features:

1. The environment facilitates the integration of software components. In this regard, the compatibility among individual components and different CASE tools is the key issue.
2. The environment is an open environment. Investigators working in related areas at the same site or at remote sites may extend the environment by

adding components and/or CASE tools to the environment.

3. The environment is domain-specific for structural engineering. However, many of its components and CASE tools are directly applicable to other engineering domains.

#### 1.3.5 Benefits

The following benefits are expected from the SESDE:

1. New substantial applications may be built more efficiently based on reuse of software components accumulated from previous software development. Rapid prototyping of new algorithms and new approaches needed for effective research can be more easily accomplished.
2. The effects of hardware, operating system, and graphics library evolution may be resolved within the SESDE system without affecting the applications.
3. The sharing of software and collaborative work among developers in the same or remote sites can be greatly facilitated.
4. New computer technologies such as computer graphics, advanced user-interfaces, and databases management can be made more readily available to researchers.

### 1.4 Components of the SESDE

#### 1.4.1 Classification

The components of the SESDE are developed as object classes. According to an object-oriented methodology, the envisioned environment is composed of the following groups of object classes:

1. **Object Sub-Systems:** Several levels of classes can be related by an inheritance mechanism and grouped together to form a sub-system which implements a high-level abstraction of a particular engineering software tool. The envisioned SESDE includes the following three sub-systems:
  - A Graphical User-Interface Development System;
  - A Database Management System;
  - An Artificial Intelligence System;
2. **Generic Object Classes for Engineering Computing:** Object classes in this group include the representation of basic mathematical entities such as matrices, vectors, tensors, and functions (the mathematical meaning rather than the programming meaning). Other classes in this group are those used for basic data structure representations and general utilities in engineering software. Examples of these include text strings, extendible arrays, and an exception-handling utility etc.
3. **Specific Object Classes for Structural Engineering Computing:** Specific sets of object classes are needed to facilitate applications software development in the structural engineering domain. Typical examples are object classes for finite element analysis.

These classes are described in the following sections.

#### 1.4.2 A Graphical User-Interface Development System

Interactive graphical user interfaces are an essential part of modern engineering software. However, the code which handles the graphical user-interface is often complex and difficult to debug and modify. It accounts for a significant portion of the code of interactive graphics applications. Therefore, the design and implementation of the user interface of a program is a very

important but difficult task. To ease the development of and to allow rapid generation and modification of graphical user interfaces, and to provide a crucial layer between applications software and the various evolving user-interface environments, a Graphical User-Interface Development Sub-System is necessary. This system, called GUIDES, is a software tool consisting of reusable software components for creating and handling the graphical user-interface of applications. The design of GUIDES (Zhang, et al. 1990) has evolved from a study of currently available user-interface tools such as the Macintosh Toolbox (Mednieks, et al. 1986), MacApp (Schmucker, 1987), HyperTalk (Shell, 1988), and the X11 toolkits (McCormack, et al. 1988).

The GUIDES system provides programmers with a reasonably complete set of user-interface tools such as menus and dialogue boxes. GUIDES has facilities similar to the emerging GUI standards such as OSF/Motif (Open Software Foundation, 1990), and it is fully integrated with a modern object-oriented, three-dimensional graphics library, HOOPS (Wiegand, 1988). The key feature of any graphical user-interface development system is the achievement of a better separation between the user-interface and other components of an application (Dodani, et al. 1989). GUIDES provides an Interface Description Language to achieve this feature. Applications can use this language to specify their user-interface independently of the application-specific code.

With such a system, researchers can concentrate their efforts on the functionality of the program at hand without getting bogged down in the details of implementing the user-interface. The complexity of the design and implementation of the graphical user-interface is significantly reduced. The code for creating and handling the user-interface is completely separated from the computational components of applications. Application-specific and user-interface components can be independently designed, developed, tested, and modified. A detailed description of GUIDES is given in PART TWO.

### 1.4.3 A Database Management System

Engineering analysis and design software systems usually need to handle a large amount of data. A typical program needs to obtain input data either interactively or from databases in the file system. It must also check the legality of the input data to ensure the correctness of the computations. The code that handles the data input is the most cumbersome and error-prone part in many programs. Furthermore, in most modern applications, the input process is substantially compressed by the use of sophisticated user-interfaces and computer graphics. The resulting data is then greatly expanded prior to performing the engineering tasks. The input data and the large amount of data created during execution of a program must be passed to and received from different code units to perform the desired operations. A program also needs to store the computational results for further processing. Often, several databases are shared by a number of systems. Thus, the enforcement of data consistency between different systems becomes an important issue in software development. In many cases, data handling is where the inflexibility and inextensibility occur.

A database management system responsible for the data transfer from the user to the program, between code units in the program, and between different programs, is thus necessary. The database management system envisioned for the SESDE is vital to the development of standardized reusable components and to the integration of reusable components into applications. With this system, application programmers will be substantially relieved from the handling of data input, data transfer between primary and secondary memory, and enforcement of data consistency between applications and application components.

During the period of the present research, a new technology, Object-Oriented Database Management Systems (ODBMS), is emerging and appearing on the commercial market. This new technology offers many advantages over the older database technologies. The development of an ODBMS involves an intensive software development effort. Thus, it is not feasible to develop an

ODBMS component for the SESDE. Rather, a commercially available ODBMS should be integrated into the SESDE. This aspect is similar to the integration of HOOPS into the SESDE for handling the basic graphics functionality. Enhancements to a DBMS will generally be required that are similar to the development of the GUIDES software which enhances the HOOPS in the aspects of handling graphical user interfaces. The discussion of the issues involved with integration of an ODBMS into the SESDE is given in Chapter 13.

#### 1.4.4 An Artificial Intelligence System

Artificial intelligence based tools are being used increasingly in domain-specific engineering applications. The tools are particularly attractive for tackling ill-structured and ill-posed problems. There is a whole range of engineering problems, ranging from analysis to design and optimization, that do not gracefully lend themselves to rigid algorithmic solutions. When integrated with graphical user interface and database management systems, an environment utilizing current developments in artificial intelligence techniques would provide a powerful research tool.

For example, the complexity of the problems typically addressed and the substantial amount of data and knowledge generated by research can provide a useful testbed for neural network applications. Furthermore, a knowledge based expert system could be used to query, maintain, update, and scrutinize the validity and reliability of engineering data and knowledge bases. A knowledge based system can also be used to aid in quickly familiarizing a user with the details of a particular application program or of a software component. This will expedite the integration of new software with existing codes. Moreover, a knowledge based system can be used to control and monitor the processes of analysis, design, redesign, and optimization, as well as help interpret results and suggest possible avenues for further action.

Artificial intelligence tools can be used in tandem with research applications to build powerful and attractive ICAI (Intelligent Computer Aided Instruction) courseware. This courseware, formulated with the help of experienced faculty, will give students greater control and flexibility in the learning process.

In the SESDE, the artificial-intelligence system will be based in part on a suitable domain-independent expert system shell providing an adequate inference and knowledge acquisition mechanism. However, the development of this system is not included in the present research.

#### 1.4.5 Object Classes for Engineering Computing

There are a number of basic mathematical entities which are commonly manipulated in engineering software. Such entities include matrices, vectors, tensors, single variable functions, (i.e.,  $y = f(x)$ ), and functions of multiple variables (i.e.,  $y = f(x_1, x_2, \dots, x_n)$ ). Manipulations on these basic entities often constitute the fundamental part of an engineering program. In traditional programming languages, these entities often are represented implicitly by variables of different built-in data types. For example, a full matrix often is represented by an array and its dimension variables, and a sparse matrix is represented by a one-dimensional array which stores the elements of the matrix, an index array, and several integer variables such as the array's dimension, band-width, etc.

By using object-oriented paradigms and object-oriented languages, these entities can be explicitly represented and manipulated by corresponding object classes in software. Each entity will be an object of a particular class and can be manipulated in a way similar to its corresponding mathematical expressions. Thus, based on these classes, operations on these mathematical entities will be coded more abstractly and expressively, and they will be less error-prone.



There are also some basic data structures and general utilities which are commonly used in engineering software. The extensible array is a typical example among the basic data structures. The array type is a built-in data structure provided in all general-purpose languages. The size of an array can not be changed once the array is created. This causes problems when the exact number of elements required is not known at the time of array creation. The extensible array is a data structure used to create arrays for any specific element type. The size of an extensible array may be automatically extended whenever it is necessary. At the same time, the elements of the array can still be accessed by using an index which is the same as the built-in array data structure. An exception-handling utility is a typical example of general utilities. When an exception occurs in an application, this utility may report the error, invoke application-specific error-handlers, and send a signal to the operating system to abort the execution of the application if necessary.

Reusable software components which implement these basic data structures and general utilities are necessary for efficient development of quality engineering applications. It is very difficult to develop such components and make them easy to use with traditional programming languages. Object-oriented methodologies and languages make the development possible and feasible.

A set of general object classes for engineering computing has been developed in the present research (Zhang, et al. 1990) as a part of the envisioned environment. These classes are described in PART THREE.

#### 1.4.6 Object Classes for Structural Engineering Computing

The components described in the previous sections are general-purpose in that they can be used readily in the development of any engineering software. A group of reusable components specifically for structural engineering computing is

an essential part of the envisioned environment.

Many different components may be utilized in any particular set of structural engineering applications. Among these are many common or similar software components. Object-oriented programming provides the means to utilize both commonality and similarity.

One example of this is a family of classes which implements different element types in a finite element analysis. A generic element class may be developed which can be used as the base class for any specific element type. A class for a specific element type would inherit the properties and methods defined in the generic class. Only the properties and methods which are specific to a particular element type would then need to be implemented for a specific element class.

Other families of classes for finite element analysis would include those for various types of constitutive models, integration algorithms, and global analysis strategies. Standard interfaces can be made for the classes in each family such that a programmer can easily integrate them into an application without knowing their implementation details.

These types of classes will form a structural engineering specific object class library to facilitate structural engineering software development. This library and the general-purpose reusable components described in previous sections can be utilized in the development of specific structural engineering applications. Any new component developed for a specific application can also be stored in the structural engineering specific class library for future reuse. As a result, this library will grow progressively as new applications are implemented. However, this development is not included in the present work.

## 1.5 Domain-Specific CASE Tools

Software components form the foundation of software reuse in the development of applications. However, if software reuse is to become a reality, CASE tools are needed to manage the software components and to assist programmers with: (1) finding and selecting reusable components, and (2) integrating reusable components and software tools with application-specific components to build an application. Several CASE tools are envisioned here, but the specific development or integration of these tools within the SESDE is not included in the present work.

Three of these envisioned CASE tools are described in the following subsections. The first two CASE tools are general purpose, that is, they can be used for software development in any specific engineering domain. The last one is structural engineering specific since knowledge of programming techniques and types of applications in the structural engineering domain is embedded in the tool. However, the framework of the tools is still general and may be adopted by other engineering areas.

### 1.5.1 A Tool for Graphical User-Interfaces

As described previously, applications can represent their graphical user-interface through the GUIDES description language. A construction tool for graphical user-interfaces may be included in the envisioned environment. This tool will allow programmers to graphically define the entire interface of an application and then automatically generate the user-interface specification. It will provide a necessary facility for rapid prototyping and incremental development of application user-interfaces.

### 1.5.2 A Tool for Reusable Component Libraries

The software libraries described previously will form databases of reusable software components. Not only should these databases contain the code corresponding to each of the software components, but also they should maintain information about each component such as a component's specification and the dependency between a component and other components. A CASE tool is necessary for proper management of these libraries. This tool will help authorized personnel manipulate and maintain the libraries. It will also provide an interactive interface for programmers to retrieve information, such as a catalogue of components or the specification of an individual component from the libraries. Furthermore, this CASE tool will also be used to search for components with specific attributes.

### 1.5.3 A Tool for Application Development

Application development processes can be automated by use of a large collection of reusable software components in a specific domain. The computer should become an active and efficient assistant for the building of applications. A computer-aided application development tool may be developed to facilitate automated software development for structural engineering.

A programmer will input the requirements of an application to this system. According to the requirements, the system will check if reusable components in the library are sufficient to construct the application. If they are, these reusable components will then be integrated by the system to form the application. If they are not, the system may inform the programmer of which components need to be developed for the application. With the help of such a tool, applications can be developed in a more efficient manner. With this type of tool, the traditional general-purpose analysis program containing many software components may not be necessary because a program containing only

the necessary components for a specific analysis can be generated directly. A program generated in such a way will be much smaller and more efficient than a general-purpose program.

### 1.6 Objective and Scope

The critical issue addressed by the present work is that efforts made on development and extension of software for engineering computing often cannot be accumulated. Rather, they become wasted. A potential solution is to improve the reusability of software components. The objective of the work is the design and development of a software development environment that promotes software reuse in the specific structural engineering domain. This environment should provide a systematic support to the development of applications software, as well as serve as a crucial layer between structural engineering applications and the evolving computer technology.

The present work attempts to build the basic framework of the SESDE, and to establish a model of such an environment for other engineering areas. Herein, the architecture and major components of this environment are identified. Requirements are established for many of the components. Several of the components are designed and implemented. The necessary technologies for the design, development or integration of other components are outlined.

Software engineering methodologies for engineering software development in general, and object-oriented programming approaches in particular are reviewed. Basic functionalities of the graphical user-interface development system are designed and implemented. The application of general database management technology to engineering software is evaluated, and the issues involved in integrating a commercial object-oriented database management system with the SESDE are investigated. A set of general object classes for engineering computing are designed and developed.

## 1.7 Organization of the Thesis

This thesis is composed of three parts and three separate chapters. This chapter is an overview of the present research.

PART ONE gives an overview of software engineering principles and technologies that will be applied in the development of the SESDE. There are four chapters in the first part. In Chapter 2, software engineering technologies are briefly reviewed. Chapter 3 discusses software reusability issues. A summary on the object-oriented programming paradigm and the use of this paradigm in the C and C++ languages are presented in Chapter 4. Lastly, Chapter 5 discusses the topics on how to apply software engineering principles and technologies in the SESDE development.

PART TWO deals with the Graphical User-Interface Development System of the SESDE (GUIDES). The current technology of the development of user-interface tools is discussed first in Chapter 6. Chapter 7 gives an overview of the development of GUIDES. A detailed description of the GUIDES is given in Chapter 8.

PART THREE describes object classes for engineering computing in general. The current state-of-the-art in engineering software development is first reviewed in Chapter 9. General object classes currently in the SESDE object library are then described in Chapters 10, 11, and 12.

Chapter 13 addresses the requirements for a Database Management System of the SESDE. Current database management technologies are reviewed. The issues of applying these technologies for engineering data management and the integration of a commercial object-oriented database management system with the SESDE are discussed.

Chapter 14 provides the concluding remarks. Relevant references are listed at the end of each corresponding part.

**PART ONE**

**OVERVIEW OF SOFTWARE ENGINEERING TECHNOLOGIES**

## CHAPTER 2 BACKGROUND AND CURRENT ISSUES

The discipline of software engineering was born in the late 1960s to overcome the so-called "software crisis" (Bishop, 1986). This crisis resulted directly from the introduction of a new generation of computer hardware. These computers were orders of magnitude more powerful than the older generation, and their power made hitherto unrealizable applications become feasible. The implementation of these applications required building large software systems. However, existing techniques which were applicable to small systems could not just be scaled up, and were inadequate for building large systems.

A number of major projects were late, unreliable, difficult to maintain, cost much more than predicted, and performed poorly. Software development was then in a crisis situation. Hardware costs were down while software costs were rising rapidly. Thus, there was an urgent need for new techniques and methodologies which allowed the complexity and costs of large software systems to be controlled, and the people involved in software development to be managed and motivated (Sommerville, 1985). The field of software engineering was born to meet these demands.

The term Software Engineering is defined as "The profession of applying scientific principles to the design, construction, and maintenance of computer software systems" (Sommerville, 1984). This definition emphasizes that software engineering addresses all stages of the life-cycle of a piece of software: specification, design, implementation, validation, operation, maintenance, extension, and reuse. At present, software engineering principles are well



established with regard to central issues such as program structure, program design technique, user-interface design, program documentation, software reuse, and software development environments. These principles have been successfully applied in the computer science profession.

This chapter presents a brief overview of software engineering technologies. In Section 2.1, the classical software life-cycle model is described. Section 2.2 discusses important characteristics of a well-engineered software. Lastly, in Section 2.3, some issues in current software engineering research are highlighted.

## 2.1 The Software Life Cycle

In the development and use of a software system, a number of distinct and interacting phases can be identified. The term software life cycle is used to describe these phases of a software system. In the traditional software engineering approach, the life cycle of a software system is broken into five major phases: specification, design, implementation, testing, and operation and maintenance.

**1. Specification:** The first stage in producing a software system is to generate a requirements specification which defines the functions to be performed by the system and the constraints on the system. The resulting software specification is not concerned with the internal operation of the system but rather with its external characteristics.

**2. Design:** A software design is a machine independent statement of how individual program units must interact to implement the software specification. The quality of the design of a piece of software affects not only the implementation of the design but also the life cycle costs of the resulting product. A well designed program will not only be more reliable and require less

maintenance than a program developed in an ad hoc manner, but also it can be more easily modified as the product requirement changes during its lifetime. A traditional design methodology used in the development of software systems is top down structural programming. In this methodology, the design is developed in a series of stages with each successive stage being a more refined version of the last until, finally, the complete system design is produced.

3. **Implementation:** The software design is realized in this phase by code written in a computer programming language which can be executed by the target computer destined to run the software system. The implementation of a software system must involve two important considerations: the programming language and the programming environment to be employed. The programming language, which is the medium through which programmers build the system, affects the ease of development and maintenance of the system. The environment in which a software system is developed is of crucial importance to the success of a project since it has a great impact upon programmer efficiency, documentation quality, project management and product quality (Depledge, 1984).

4. **Validation:** This phase involves the validation that the implemented software meets the requirements of the user. To increase overall confidence in the software, it is necessary to perform tests to identify and correct errors introduced in the system during the implementation phase. Moreover, during this phase, it is common to detect errors and misunderstandings in the functionality of the system introduced in the preceding phases.

5. **Operation and Maintenance:** After the software is in use, it may be necessary to correct errors detected or to modify the software to meet changes in user requirements. This phase is also called "software evolution". Overly complex program structures are generally the result of evolutionary development rather than a single creative act. This is especially true for large software systems.

It should be noted that the software life cycle is a cyclic rather than a sequential process where each phase interacts with preceding and succeeding phases. Usually, work done in an early phase of the cycle must be redone as problems arise in later phases. Costs of software development are not equally distributed in these phases. It is estimated (Sommerville, 1984) that the first three phases, specification, design, and implementation, each accounts for about 20% of the total initial costs with the remainder taken up by validation. Typically, the maintenance cost is much higher -- about five times the development costs.

## 2.2 Characteristics of Well-Engineered Software

According to the cost distribution over the life cycle, a well-engineered software should exhibit three basic and dominant characteristics (Sommerville, 1984):

1. It should provide the functionality and operate within the constraints defined in the software specification.
2. It should be reliable.
3. It should be readily modifiable and extendible.

The first is a very general characteristic which includes many external features that may be defined in the software specification, such as space and time efficiency, and ease of learning and use. The second is the most important dynamic characteristic of a software system as software becomes more diverse and is used in more and more application areas.

The third characteristic arises directly from the consideration of the maintenance costs. The software development activity should be aimed toward producing a readily maintainable and readily extendible software system. This

means that the software system must be constructed so that modification and extension can be accomplished in a time proportional to the magnitude of the changes rather than to the size of the system. This characteristic is critically important for software systems to survive in an evolving environment. Thus, it should be a primary goal in software development. This characteristic leads to several important issues of software engineering including program structure, design techniques and reuse of program components, program readability, and program documentation.

Maintainable systems should have a modular structure both in the small, to allow modification on the functionality of a specific component, and in the large to allow changes in major components as the application domain as well as the computer technology evolves. Different design methodologies may lead to different definitions of modules. However, the modules in a software system should be independent of other modules to a certain extent (i.e., they should be loosely connected). Moreover, the interface between modules should be clearly defined such that the modules in a system can be independently developed and tested, unconnected or replaced without side effect, and kept in libraries for reuse. The readability of programs should take place over writeability because programs are read more often than they are written. Documentation should be clear, concise, and complete, and should be treated as a part of the software system of equal importance to the actual source code.

### 2.3 Current Issues

It is likely that the average size and complexity of future engineering software will grow considerably in the coming years because of: (1) increased demand for sophisticated user interfaces and graphics processing; and (2) rapid development of hardware which makes it possible to develop programs for problems which were previously considered infeasible. Thus, the improvement

of software quality and productivity is becoming increasingly important. This is an issue in software engineering research that has attracted increasing attention during recent years (Wegner, 1984; Barbacci et al., 1985; Prieto-Diaz et al., 1987; Meyer, 1987, 1988; Tracz, 1987; Burton et al., 1987, Pyster et al., 1988; Ellison, 1988).

At present, software development is still largely labor-intensive -- programmers perform the major activities in the process. Only by becoming more technology-intensive can software quality and productivity be improved on a large scale. In technology-intensive software development, the computers become more important actors. Automation in software development can be accomplished by using: (1) reusable software components and software tools that individually address a single area or function; and (2) development environments that consist of large collections of reusable components and software tools as well as associated programming tools which facilitate software reuse.

The objective of software reusability is to enhance the software development process by enabling effective reuse of software components, designs, templates, or specifications to substantially increase the fraction of a new system that can be derived from prior work. Among different types of software reuse, the reuse of software components has the most direct and tangible benefit. Software component reuse requires a software design methodology which promotes reusability. This has made object-oriented programming a popular choice in recent years.

Reusability also requires a rich software development environment in which software reuse is systematically supported and is a natural activity. An ideal integrated environment should be composed of: (1) a consistent and compatible set of components and software tools, and (2) integrated programming tools for all phases of system development and operation. The consistency of reusable components must be emphasized. The quantity and

merits of available individual reusable components in an environment are less important than the consistency of these components. Consistency means not only the ability to communicate with each other, but also the ability to share a common set of calling conventions, user-interfaces, and general design philosophy (Barbacci et al., 1985). These issues will be discussed further in the following chapters.

## CHAPTER 3 SOFTWARE REUSABILITY

Because of the potential benefits of large scale software reuse in software development and maintenance, software reusability has attracted increasing attention over the past years and is one of the major interests in software engineering research and practice (Wegner, 1984; Barbacci, et al., 1985; Prieto-Diaz et al., 1987; Meyer, 1987, 1988; Tracz, 1987; Bassett, 1987; Kaiser et al., 1987; Burton et al., 1987; Lenz et al., 1987; Fischer, 1987; Love, 1988; Pyster et al., 1988; Cox, 1988). This chapter presents a brief overview of the issues related to software reuse.

### 3.1 The Concept of Reuse

Wegner (1984) has made an interesting comparison between software technology and the technology that fueled the industrial revolution. He points out that software technology was labor-intensive in its youth and is becoming capital-intensive as it matures. Capital-intensive development can be referred to also as being technology-intensive. An important feature of technology-intensive software development is software reuse.

In fact, reusability is a general engineering principle whose importance derives from the desire to avoid duplication and to capture commonality among inherently similar tasks. Designing software without reusing existing tools and/or software components is similar to designing a building without using prefabricated structural components and standard member section sizes.

Reuse may be defined as the utilization of previously acquired concepts and objects in a new situation. It is a matching process between new and old situations and may take place at different stages of software development. A model of a real world object may be reused in different software systems again and again. The specification or design of a finite element analysis program may be reused for the development of another finite element program that has similar functionality with the previous one. The code of a software component in a program may be reused in another program without or with only minor changes. However, only the reuse of software components has the most tangible and direct benefits.

Software components can be reused by

1. being included in a variety of applications,
2. being utilized in successive versions of a given program, or
3. being called repeatedly during program execution.

All of these three forms of reusability reduce the efforts of software development. The first two forms motivate the development of reusable software components as basic building blocks for various applications. The third form motivates the development of generic software tools, such as tools for handling the graphical user-interface.

It is a mistake to assume that software component reuse does not pose any new design challenges (Fischer, 1987). Rarely is it feasible to decompose an existing software system into reusable components that can be employed readily for constructing other systems. Software components must be specially designed for reusability to achieve optimum reuse. The need for reuse has caused an evolution of the software life-cycle model and program design process. A software environment that supports reuse is essential to allow programmers to take advantage of previous work. These issues are discussed in the forthcoming.



### 3.2 Characteristics of Reusable Components

Reusable software components, or reusable modules, are basic units for building applications. A reusable component should consist of two separate parts: the specification and the implementation. The specification should consist of three major parts: (1) an overview of the component, (2) the definition of the component's interface with its clients (or other software components that it can communicate with), and (3) test cases for the component. The implementation of a component includes the detailed design and code implementation.

According to the information hiding principle of software engineering, the details of a component's implementation should be hidden from the outside world. Any changes in the implementation should not affect its specification. Users of a component should only need to access the component's specification without referring to its implementation.

The definition of the interface in the specification describes the available operations by and only by which clients communicate with the component. Both the syntactic interface and the semantic interface have to be defined. The syntactic interface specifies compile-time invariants that determine how components fit together, while the semantic interface specifies execution-time invariants which determine what data the component operates on.

The interface should be unified among the components of the same type (e.g., components that implement different strategies for solution of linear simultaneous equations, or components which implement different material models in a nonlinear finite element analysis). Only by a unified interface can a reusable component be replaced or added to a program without affecting other components.

The simplest possible notion of software reuse is the "use-as-is" notion (Bassett, 1987). Program development may benefit from fixed, use-as-is components. However, it is more often the case that a software component is

needed which is similar to an existing component (Meyer, 1987). Therefore, the "same-as-except" notion ("A" is the same as "B" except ....) is a needed generalization of the "use-as-is" notion (Bassett, 1987).

In this regard, extendibility, which is defined as the ease with which a software component can be modified to reflect changes in its specification, should be another important characteristic of reusable components. The UNIX operating system provides a good example of software extendibility. Existing programs in UNIX are treated as independent software components. If one of these software components does not meet a specific need, it may not be necessary to change the component itself. Either the pipe facility may be used to link the component with another component, or specific program code may be developed and placed around the component to provide the needed functionality.

### 3.3 Design of Reusable Components

Reusable software components for a certain problem domain can be identified and defined by a domain analysis. Abstraction techniques are the most powerful tool to perform this analysis. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences (Wegner, 1984). Many different abstraction mechanisms have been proposed as a basis to identify and define software components. Each may represent different types of components from which programs can be constructed, and each may result in different paradigms or methodologies for programming. Two important abstraction techniques are functional abstraction and data abstraction.

Functional and data abstraction emphasize different aspects of software reusability. Functional abstraction emphasizes reusability of functions, and

functions or subroutines are the basic reusable units. Data abstraction emphasizes reusability of types of data objects for various operations that may be applied to them. Data types or classes of data objects are the basic reusable components. Functional abstraction leads to top-down structured programming. Data abstraction, incorporated with the information hiding principle, leads to object-oriented programming.

### 3.3.1 Functional Abstraction

Functional abstraction may be specified in terms of input-output relations of a component. Every input  $x$  determines a unique output  $f(x)$ . The output depends only on the input  $x$  and on no other data. A client of the component is aware only of the input-output specification and not of the way the function is implemented. The specification represents the interface with the client, and the implementation is hidden from the client. In this approach, reusable components are subroutines.

Building subroutine libraries is a classical technique. Subroutine libraries have significantly affected the production of mathematical software systems as well as software for string manipulation and I/O. Each routine in a library implements a well-defined operation. However, library subroutines are not sufficient to achieve a large-scale improvement in software productivity and maintenance. An individual subroutine is too small and the effort necessary to make many subroutines work together is too large. Small size subroutines are more amenable to reuse than large units, since they tend to be relatively simple and context-free, but only a small amount of code is then actually reused by each subroutine call (Kaiser et al., 1987).

Moreover, in dealing with a complicated problem with many different special cases, either a single routine for all special cases or a set of routines, each corresponding to a special case, may be developed. A single routine will need

many parameters, and will probably be constructed using a set of *case* instructions leading to a complex and inefficient implementation. The addition of a new case will mean modification and recompilation of the entire module. This may introduce new bugs into the system. A set of routines will be large, and it will consist of many routines that in fact are very similar. The key problem is that there is no simple way by this approach to utilize this similarity between these routines. Client programmers have to find their way through a maze of routines (Meyer, 1987). Also, subroutines are written with all the details filled in. Therefore, it is not possible to extend the algorithm encapsulated in the subroutine without a proliferation of different versions of the code (Kaiser et al., 1987). This results in difficulties in extending an existing component.

### 3.3.2 Data Abstraction

In data abstraction, the information which should be hidden from the user includes the data as well as implementation of the functions that operate on the data. Data abstractions have an internal state that "remembers" the effect of past operations and allows components to use this state to direct future operations. Thus, the output  $f(x, s)$  from an operation of a data abstraction depends not only the input  $x$ , but also the hidden state variable  $s$ . A data type or an object class is an implementation of a data abstraction. A data type defines a common data structure which is shared by data objects of that type, and supports operations that operate on the data objects. This approach leads to object-oriented programming. The reusable components are data types or object classes.

The term "object" is used to denote software components that have a hidden state and a set of operations for transforming the state. Objects package together both the functions (called the methods of the object) and the particular

type of data that the functions are designed to work with (called the properties of the object). Objects of a same class share common type of properties and have common methods. The definition of the particular data type and the implementation of the methods of a class are encapsulated (i.e., hidden) in the class body. The methods, which are the only means for manipulating an object of that class from the outside world, are declared in the specification of the class.

Object-oriented programming has two distinguishing features which encourage software reuse:

1. The user of an object class is more clearly separated from the developer. Users are not aware of how an object is stored or how an operation on the object is implemented. They can only manipulate an object using the methods provided by the developer. This ensures that the implementation of the encapsulated object can be changed without affecting its application.
2. In an object-oriented system, a set of object classes can be organized hierarchically by the base-derived relationships between object classes and the inheritance mechanism between a base class with its derived classes.

A derived class is a specialization or an extension of its base class. The derived class may possess the properties defined in its base class as well as additional properties special to itself. Objects of a derived class may be manipulated by the methods defined in the base class as well as those defined in the derived class. This inheritance mechanism mirrors the growth of human knowledge by building on what already exists rather than by starting from the beginning for every software component. Thus, an object class that is almost like an existing one can be created by simply specifying additional properties and methods and reusing properties and methods that the new class shares with the existing base class. Extendibility of software components is thus facilitated. Object-oriented techniques are discussed in more detail in the next chapter.

### 3.4 Software Reuse and Life Cycle Models

Software reuse also plays a central role in the evolution of the software development life-cycle model. The classical waterfall model has been described in Chapter 2. This model was developed in late 1960's. In this model, the software development proceeds through a number of stages: specification, design, implementation, validation, and operation and maintenance. However, in spite of its success, the waterfall model has some drawbacks (Wegner, 1984). First, it is geared to program development by humans rather than by the computer. This reflects the fact that the potential of the computer as an active partner in program development was not fully understood when the model was established. Another important drawback is that the waterfall model does not provide feedback concerning specification and design behavior until late in the implementation and validation phases.

The concept of software reuse has motivated a new software development life-cycle model called the operation model (Wegner, 1984; Thomas, 1989). Rapid prototyping that reuses existing software components is central to this new approach. The stages in an operational software development life-cycle are: executable specification (rapid prototype), refinement, and efficient implementation. In such a development, a display prototype is first developed to perform initial user-interface testing. This is followed by the development of a full simulation that is an executable specification of the final product. The executable specification provides early feedback to both the end-user and the system designer on the functionality of the intended system. The specification is then refined and optimized to build the final product. It is obvious that in this approach, the existence of a large amount of reusable software components is essential for building the prototype as well as the full simulation.

One other software development life-cycle model is the knowledge-based model. This model depends also on the existence of a large amount of reusable software components in a certain problem domain. Knowledge of the general

programming process as well as the programming process in a specific problem domain is needed to implement this model. This model is based on the operation model, and has the potential of increasing software productivity by several orders of magnitude (Wegner, 1984). It is a specialization of the artificial intelligence technology applied to the domain of program development. This model addresses the automation of software development, and the computer becomes an active partner in the development.

### 3.5 Reuse of Components

The object-oriented design process, which is based on utilization of reusable software components, is different from the traditional process of top-down design and step-wise refinement. An object-oriented design begins with the identification and classification of the objects that the application must manipulate. The objects or classes of objects identified are then compared with the existing objects or classes which have been coded in the existing reusable software components to see if the same or similar objects or classes exist.

Existing objects or classes of objects can be used as is, or they can be extended using the inheritance mechanism. Extension requires development of new software components based on existing ones. Objects or classes which can not be found in existing objects are created as new objects or new classes, and the corresponding software components must be developed. The newly created objects or classes can be kept for future reuse. These objects or classes are then combined to form the final system.

This process is not top-down. Rather, it is a combination of top-down and bottom-up processes. Top-down design may be most effective for developing individual algorithms and routines. However, it is inappropriate at the system design level because it promotes one-of-a-kind development rather than the development of general purpose reusable software components (Meyer, 1987).

### 3.6 Support of Reuse

Optimum software reuse requires a sophisticated software development environments. Software development environments have attracted considerable attention over the past five years (Barbacci et al., 1985; Bishop, 1986; Burton et al., 1987; Fischer, 1987; Rappaport et al., 1988; Ellison, 1988; Cox, 1988). A software development environment is a compatible set of tools based on a methodology for different phases of system development and operation. It supports both technical and management activities (Bishop, 1986). The environment in which a software system is developed is of crucial importance to the success of a software project since the environment has a great impact upon programming efficiency, documentation quality, project management, and product quality.

To let users take full advantage of previous work, software environments must support software reuse. Software environments must support design methods whose main activity is not only generating new programs but also maintaining, integrating, modifying, and explaining existing ones (Fischer, 1987). Software development environments supporting software reuse generally should be domain-specific. A domain-specific environment consists of a large collection of reusable components for a specific problem domain, and the knowledge of the programming process in the domain has to be embedded in the programming tools of the environment. Barbacci et al (1985) have given several possible characteristics of such a software development environment:

- It should be open and integrated -- open so that people other than the developers can add components, and integrated so that the components work together with a uniform external interface and style of doing business;
- It should have a common communication medium for integrating diverse components into systems;



- It should be incrementally developed with working interim versions available at an early date;
- It should be interactive to shorten the software development process;
- It should be evolvable so that tool changes do not make the code of the existing applications developed in the environment obsolete;
- It should be learnable and usable with a clean, easy to use, and self-documenting user interface.

Burton et al. (1987) have developed a Reusable Software Library (RSL) which is an excellent example of a software development environment addressing software reuse. The RSL is comprised of the RSL database and four subsystems: (1) Library management, (2) User query, (3) Software component retrieval and evaluation (Score), and (4) Software Computer-Aided Design (SoftCad). The foundation of the RSL is a database which stores the attributes of every reusable software component. The library management subsystem provides a set of tools to help the RSL librarian and quality-assurance personnel manipulate and maintain a software library.

The user query facility provides an interface for users to search for components with specific attributes and to generate reports about their attributes. Score helps the user select the most appropriate software to reuse by identifying components that perform the functions requested by the user and comparing their attributes to other requirements. SoftCad is a graphical design and documentation tool that has been integrated with the RSL prototype to aid the user in the high-level design of software systems. With SoftCad, a designer can develop a program architecture design by drawing object-oriented design diagrams that are interpreted by SoftCad and automatically translated into the Ada design description language.

## CHAPTER 4 OBJECT-ORIENTED PROGRAMMING

Because of its potential to achieve a high degree of reusability and extendibility of software, object-oriented programming has become increasingly popular in recent years. Object-oriented programming has been successfully applied in areas of graphical user-interface and operating systems, as well as many other diverse application areas.

The object-oriented programming paradigm models real world entities in a specific application domain directly and naturally as software objects. Object-oriented languages support the major features or characteristics of object-oriented programming. Several object-oriented languages have been developed in the recent years such as Smalltalk (Pinson et al., 1988), C++ (Stroustrup, 1987), Eiffel (Meyer, 1988), and Objective-C (Cox, 1986).

In this chapter, the characteristics of object-oriented programming are briefly reviewed first in Section 4.1. The use of this paradigm in the C and C++ languages is discussed in Sections 4.2 and 4.3 respectively. The application of this methodology in engineering software development is then discussed in Section 4.4.

### 4.1 Characteristics

There are five important characteristics of object-oriented programming (Pinson et al., 1988; and Thomas, 1989): abstraction, encapsulation, inheritance, polymorphism, and composition. These characteristics are discussed below.

#### 4.1.1 Abstraction

Abstraction forms the foundation of all object-oriented development. The intent of an object is to represent a problem domain entity. The concept of abstraction deals with how an object represents this representation of the entity to other objects. This representation is a simplified description, or a specification of the entity that emphasizes some of the entity's details while suppressing others. The stronger the abstraction of an object, the more details are suppressed.

It is a common practice that the first task in the development of an application is to decompose it into a set of abstractions which represent the entities in the real world that the software attempts to simulate. In the approach of traditional programming, an abstraction is then implemented as a set of variables of data types provided by the implementation language. Thus, the original abstractions disappear upon moving from the design phase to the implementation phase.

For example, the abstraction of a sparse matrix (such as the stiffness matrix in a finite element analysis) is often represented as an array which stores the matrix coefficients, an array which stores the information about the non-zero coefficients distribution in the sparse matrix, and several scalar variables which store information such as dimension and states of the matrix. Operations on the sparse matrix are expressed in terms of operations on these separate variables. Stress and strain tensors, as another example, are often represented as a one-dimensional array. The correspondence of tensor components with the array entries is artificially specified, e.g., the array entry  $s(4)$  may represent the tensor component  $s_{12}$ . This representation has to be understood to manipulate a tensor in a program.

In object-oriented programming, the same abstractions may be preserved throughout the design and implementation phases. An abstraction may be

represented explicitly by an object class which mirrors the behavior of the real world entity. The definition of the class contains the complete information about the abstraction and all the possible operations associated with this abstraction. An instance of the abstraction is thus represented as an object of the specific class in the code.

Using the same examples as mentioned above, an object class for sparse matrices can be defined of which the stiffness matrix is an instance. Operations on the stiffness matrix are explicitly expressed in the actual code as operations with respect to the stiffness matrix instead of operations on some separate variables which are part of the stiffness matrix representation. An object class can also be defined to represent tensors. A component of a tensor in the code can then be expressed in a similar way as in the mathematical notation, regardless the actual representation and storage of the tensor in computer. For example, a tensor component  $s_{12}$  can be expressed as  $s(1, 2)$ . If the class is implemented in an object-oriented language, operators can be overloaded with the tensor class such that code dealing with tensor operations will be more expressive. For example, the addition of two tensors  $a_{ij} + b_{ij}$  can be coded as  $a + b$ .

The benefits of preserving abstractions in software development are obvious:

1. The code is more abstract, expressive, and understandable to non-developers because abstractions of real world entities are represented and manipulated explicitly in the code;
2. The software component which contains the implementation of the object classes representing abstractions will be more reusable because the functionalities of software in a specific domain may not be the same, but the categories of abstractions in the same domain are more or less the same;

3. The component will be more easily reused due to its high coherent nature.

The decomposition of an application into objects based on the abstraction principle proceeds in the following manner:

- Identification of the object classes and their properties;
- Identification of the operations performed by and required of each class;
- Establishing the relationships between object classes by inheritance and composition mechanisms;
- Establishing the interface of each object class.

However, following the steps listed above is not an automatic procedure. It requires a great deal of knowledge about the application domain and good understanding of the particular application.

#### 4.1.2 Encapsulation

Encapsulation is the technical name for information hiding. The information hiding principle states that the implementation details of an object class should be kept secret from other classes. Objects are encapsulations of abstractions (Pinson et al., 1988). They encapsulate a set of methods and properties which are operated on by the set of operations.

Encapsulation together with abstraction separates the representation of an object class from its implementation details. Thus, it leads to two views of an object class: the outside view and the inside view. The outside view is the view by the users of the class. It captures the abstract or external behavior of objects of the class. By seeing the outside view only, one can use an object class without knowing how the class is implemented. The inside view is the view by the implementer. It focuses on the implementation of the behavior which is

encapsulated in the object class. The benefits of separating the two views is obvious. The implementation of the class can be modified or replaced without affecting the outside view of the class, and therefore, without affecting other classes which interact with the objects of that class.

Consider the same sparse matrix example again. The implementation details of the sparse matrix class, such as the storage architecture of the matrix elements, are hidden from its users. The class can be used to create sparse matrix objects and to operate on these objects without knowing whether the class implements the active column storage scheme or the constant band scheme. Thus the class' use will not be affected if the storage scheme of the class is changed.

#### 4.1.3 Polymorphism

One important goal of abstraction in decomposing an application system is to control the complexity. Controlling of complexity often leads to a hierarchy of abstractions. An abstraction in a lower level provides more detailed explanation for the behavior that appears in a higher level (Shaw, 1987). Each abstraction corresponds to a class of objects in object-oriented programming. Polymorphism and inheritance (which is discussed in the next section) are two mechanisms of object-oriented programming that utilize the commonalities or similarities among classes of objects.

Polymorphism is defined as the ability of different objects to respond differently to the same messages. Here, the point is that if different classes of objects respond to the same messages, they can be treated identically, regardless of how they respond to these messages. Each class of objects should respond to the same messages in ways appropriate to the kind of object class that it is. The details of how an object class responds to messages are hidden from the outside world.

From another viewpoint, polymorphism also states that a standard interface should be defined for object classes which are similar or which are at the same level of an abstraction hierarchy. Thus, objects of similar classes can be treated identically. As a result, it becomes easy to: (1) insert a new class of objects to a system if the new class shares the same interface as the existing classes; or (2) replace a class by another which shares the same interface as the one to be replaced. Polymorphism leads to a style of programming referred to as differential programming or programming by modification (Thomas, 1989).

For example, several classes for matrix manipulation are developed in the C++ language in the present work (Zhang et al., 1990c). Each class represents a type of matrix which has a certain characteristic such as symmetry or diagonalness. The operation function *product*, which multiplies a matrix by a scalar is defined in different matrix classes with the same function specification. Thus, such an operation can be coded identically in an application by calling the function *product* through the matrix object being operated on, regardless of the actual class of the matrix object. As another example, a standard interface may be defined for several sparse matrix classes. Each of these classes implements a specific sparse matrix storage scheme. Because of the standard interface, the sparse scheme used in an application can be changed without affecting the other components in the application.

#### 4.1.4 Inheritance

The other characteristic of object-oriented programming that supports differential programming is inheritance. Inheritance is the ability to define a new object class that is just like an old one except for a few minor differences. A class is called the base class of any class which is immediately under it in the class hierarchy. A class is called a derived class of its base class. An object class inherits the properties and methods from its base class as well as from all of its

ancestor classes (i.e., all of the classes which are above it in the class hierarchy). Because a derived class is a more refined specialization of the base class, a derived class can define new properties and methods for its objects. Thus, an object of a class can be operated on by the methods defined in all its ancestor classes as well as the methods defined in its own class. The code implemented for a class is reused by its derived classes. This is the most important characteristic that distinguishes object-oriented programming from other programming paradigms.

Consider the matrix classes mentioned previously. Different matrix abstractions such as a symmetric matrix, a diagonal matrix, and a lower-triangle matrix are specializations of a high-level abstraction, a general matrix. These abstractions are represented respectively by different matrix classes such as the class *Matrix* for general matrix abstraction, *SMatrix* for symmetric matrix abstraction, *DMatrix* for diagonal matrix abstraction, etc. The generality and speciality between these matrix abstractions is utilized in the matrix classes by defining the class *Matrix* as the base class for the other matrix classes. The declaration of the *Matrix* class contains the common information for matrix representation such as the dimension of the matrices. Operations are also defined in the *Matrix* class which perform the operations applicable to objects of any specific matrix class. A derived class of *Matrix* such as the *SMatrix* or *DMatrix* classes only implements operations where advantage can be taken of the characteristics of the special matrix abstraction it represents.

Class inheritance may be used not only for managing hierarchical relationships among object classes, but also for system evolution and incremental modification. A new class of objects can be easily created from an object class which is similar to the new class by adding more specific details. The use of inheritance to specify incremental change flexibly is invaluable in software engineering (Wegner, 1989).



Both the characteristics of polymorphism and inheritance are based on commonalities or generalities among classes of objects. Finding commonality among objects in a system is not a trivial process. How much commonality can be exploited depends on how the system is designed. Commonality or generality must be actively sought when the system is designed, both by designing classes specifically as building blocks for other classes and by examining classes to see if they have similarities that can be exploited in a common base class (Stroustrup, 1988a).

It is clear that not everything can be organized into a single inheritance tree. Single-inheritance systems require that classes are organized into a tree structure. This can sometimes result in deep inheritance structure that can be awkward to use (Thomas, 1989). An alternative is based on multiple inheritance which is supported in several object-oriented languages (C++ supports this).

#### 4.1.5 Composition

Many entities in the real world are often complicated. A complex entity may contain many entities of other types. This fact is simulated in object-oriented programming by the composition mechanism. The composition mechanism states that a complex object may be assembled from objects of other classes. Thus, the code implemented in the classes of the component objects can be reused to accomplish the functionality of the complex class. In this way, composition is another key to reusability (Thomas, 1989).

For example, exception handling is an important feature for many object classes. When an exception occurs, an object class usually should report the exception by delivering an error message and sending a signal to the application to let the application make the final decision on what to do with the exception. Such a functionality is not hard to implement, but it is cumbersome to include the same code for exception handling in every object class. Thus, a class can be

developed specifically for exception handling. An objects of this class can then be used in another class as a component of that class for exception handling.

It should be noted that inheritance is a mechanism to create object classes that share properties and methods with similar classes, while composition is a different mechanism that allows assembling of composite objects. Thus, in an object-oriented system, there are often two hierarchical structures, the inheritance hierarchy, and the composition hierarchy. It is easy to confuse the two hierarchies. The basic difference between the two is that inheritance is related to the relationships between the object classes, while composition is the relationship between the objects, or the instances of different classes. An object of a composite object class is built using instances of other classes.

#### 4.2 Programming in the C Language

Object-oriented programming (OOP) is simply a paradigm or a style. It can be performed with any general-purpose programming language to a certain degree. Nevertheless, as discussed by Stroustrup (1988a), there is an important distinction between a language which supports OOP and one which merely enables its use. A language supports the OOP paradigm if it provides facilities that make the use of OOP convenient, safe, and efficient. A language does not support OOP if it takes exceptional effort or skill to write programs in which the paradigm is utilized. In such a case, the language merely enables the use of the OOP paradigm.

Stroustrup also explains that support for OOP paradigm comes not only in the obvious form of language facilities, but also in the more subtle forms of compile time and run time checks for unintentional deviations from the paradigm (1988a). Thus, if a language is employed which supports OOP, artificial strict rules on low-level programming details are unnecessary because these rules will be enforced by the compiler.

It would then appear that use of an object-oriented language is desirable for object-oriented software development. However, performing OOP with a classical procedural language may still be a practical and viable approach for many organizations (Coad et al., 1990). This section discusses techniques for OOP in the C language. The C language is one of the most widely used languages in the world today (Kernighan, et al., 1988), and it is the major language for many software development organizations. In addition to its key strengths of flexibility, efficiency, availability, and portability, the C language also provides relatively rich syntactic features that make implementation of OOP paradigm easier to accomplish than with other procedural languages such as FORTRAN. However, since C does not support OOP, programmers must follow strict coding disciplines to code object-oriented features explicitly.

Although the use of C for object-oriented software development is debatable, there are some reasons for following this approach at the present time:

- Most object-oriented languages are still in their youth. C is a mature language and it is more readily available.
- C has a more mature and more readily available programming environment. Many tools are available in UNIX and other operating systems for programming in C. In some cases, these tools may not work well with object-oriented languages such as C++.

In the present research, both C and C++ are used for the development of reusable components. The use of the C language in the present research follows the reasons listed above, and also a C++ compiler was not yet available until late stage of the present work. Moreover, the author feels that study the techniques of performing OOP in C is helpful to gain a more clear understanding of the object-oriented paradigm.

There are two major issues involved with object-oriented programming in C: representation of objects and implementation of message passing. Several techniques concerning these two issues have been discussed by Linowes (1988), Bailey (1989), and Meyer (1988). In the following sub-sections, these issues are discussed with regard to these techniques as well as the approach used in the present work for the development of GUIDES. A general utility for object-oriented programming in C, which is named CLOOP, is then presented.

#### 4.2.1 Representation of Object Classes

##### 4.2.1.1 Structure Types for Object Classes

To represent an object class in the C language, it is a common practice to define a data structure which contains the definition of the properties of the class. Each field in the structure corresponds to an item of the properties. The data structure can then be further defined as a structure type by the *typedef* instruction. Thus, an object of the class can be declared and manipulated as a whole in the code. The definition of the structure type may be placed in a header file. This header file should be included in the C source file which contains the implementation of this class and, if necessary, in the source files which contain the implementation of the derived classes of this class. Because a C file forms a boundary between different code units in a program, this structure type is unknown to the code units contained in other files where the header file is not included. Thus, encapsulation of object classes can be accomplished.

However, the structure type of the class has to be known by all the classes which communicate with this class. To this end, this header file has to be included in the source files containing implementations of other classes. In doing so, the possibility of undisciplined access to the fields in the class' data structure is risked. An alternative way to accomplish communication between classes is to introduce a general type of pointer to objects which bypasses the protection

boundary of source files. An object sends a message to another object via a pointer of general type. The pointer is then casted as a pointer to the class structure type in the methods of the class of the object which receives the message. However, it is not possible to hide some of the fields while exposing the others. All the fields of a class' data structure can only be either exposed or hidden together.

#### 4.2.1.2 Use of Nested File Inclusion for Property Inheritance

A nested file inclusion mechanism is suggested by Linowes (1988) for implementation of property inheritance. An object class is implemented in two source files: *class.c* and *class.p*, where *class* is the name of a class. The *.c* file contains the methods and message-handler of a class, and the *.p* file contains definitions of the properties of the class. The *.p* file of a class includes the *.p* file of its base class, and is included in the definition of the structure type of the class. The class' structure type is contained in the *.c* file of the class. Figure 4.1 illustrates this approach where a class *rectangle* is derived from a class *shape* which is in turn derived from a class *common*. By use of the nested file inclusion, a derived class possesses all the properties of its ancestor classes. However, a base class exposes the definition of its data structure to all of its derived classes. Strict programming discipline is needed here to not permit a class to access the properties of its ancestor classes.

#### 4.2.1.3 Use of Linked Objects for Property Inheritance

The mechanism used in the implementation of property inheritance in GUIDES (Zhang et al., 1990b) is not satisfactory either. In this design, an object of a derived class is represented by several objects in memory. These

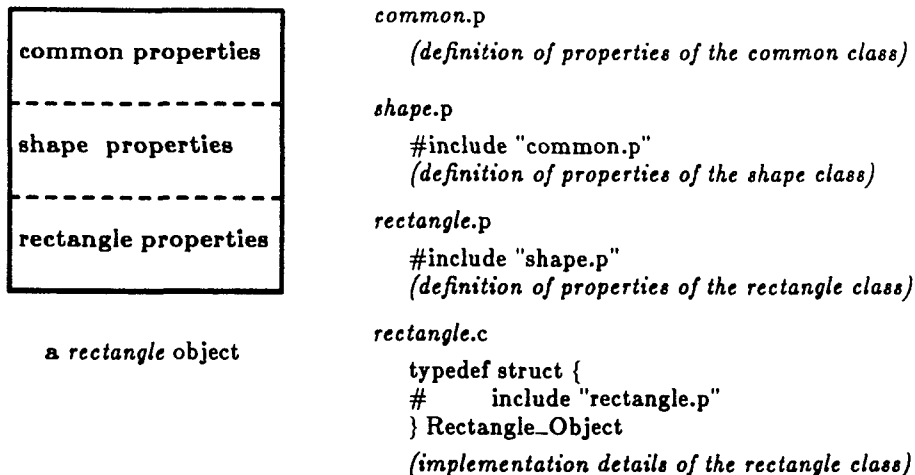


Figure 4.1 Linowes' approach for property inheritance (from Linowes, 1988)

objects are linked together by pointers. Figure 4.2 illustrates the memory organization by this mechanism for an object of a class *B*, which is derived from a class *A* which is in turn derived from the *Basic* class.

The *Basic* class is the root of the class inheritance hierarchy of GUIDES. The data structure of the *Basic* class contains properties which are common to all classes such as the class identifier of an object. It also contains a field, represented by the name *p\_derived*, which references an instance of a derived class' data structure. The type of this field is *Pointer* (i.e., a general pointer type) to bypass the protection boundary of source files as described previously. A class derived from the *Basic* class contains properties which are specific to the derived class. It may also contain a field, again represented by the name *p\_derived* which references to an instance of any class' data structure that is derived from this class. Any object in the system can then be represented as a *Basic* class type. The same problem is experienced by this approach as in Linowes' approach. The definition of the structure type of a class is exposed to

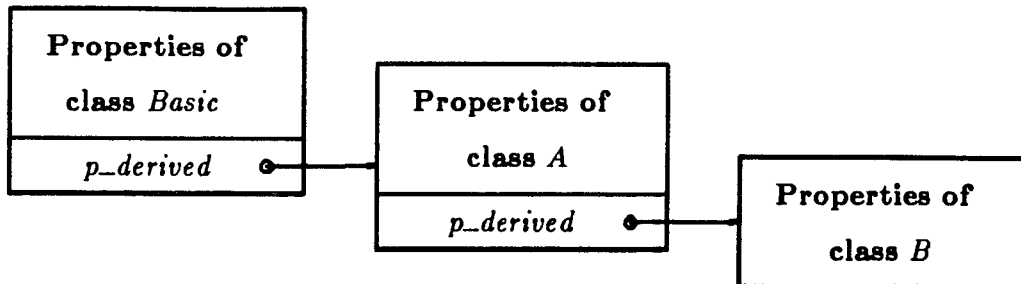


Figure 4.2 GUIDES' approach for property inheritance

the classes derived from it. However, this approach avoids the possibility of unintentionally accessing a field in the data structure of an ancestor class from a derived class.

To "support" object-oriented programming in C, it is necessary to establish a more general mechanism for encapsulation and property inheritance. The mechanism should be flexible such that a programmer can choose either to expose or to hide the definition of the structure type of a class to its derived classes as desired. Such a mechanism is implemented in CLOOP and is discussed in Section 4.2.3.

#### 4.2.2 Implementation of Message Passing

Object-oriented programming is largely a style implemented by sending messages to objects. The operation functions of the receiver objects are invoked by the messages. This is the key aspect for object-oriented programming, and it needs to be coded explicitly in C.

Polymorphism requires that an object should be able to send a message to another object without necessarily knowing the class of that object or the method which will be invoked. For example, an object should be able to send a message *Draw* to another object without knowing whether the object is a circle, a rectangle, or a curve, and without knowing the name of the method which will be invoked to perform the operation. In the case of method inheritance, sending a message to an object may invoke a method which is not implemented by the class of that object. Instead, the method may be implemented by one of its ancestor classes. Because message passing is the major means of communication between objects, the efficiency of its implementation is essential to the overall run-time performance of an object-oriented system.

Here, the problem is how to find efficiently a pointer to an appropriate operation function for a given object and message pair. The exact class type of the object may only be known at run time. Several different approaches have been suggested (Linowes, 1988; Meyer, 1988; Bailey, 1989; Zhang et al., 1990b)

#### 4.2.2.1 Use of the Switch Statement

In the approach suggested by Linowes (1988), each object class implements its own message-handler function. The handler uses a *switch* statement with cases for each message recognized by the class. It calls the corresponding operation functions upon receiving messages. These operation functions or methods of the class are defined as *static* and are located in the same source file as the message-handler. To implement polymorphism, the first field in the data structure of every object class is a pointer to the message-handler of that class. This field is defined in the file *common.p* of the class *common* as shown in Figure 4.1, and is inherited by classes derived from it.



A general message-passing function *Send*, which is a macro function, is then defined which takes advantage of the fact that the first field in the data structure of every object is the message-handler of its class. Figure 4.3 shows the implementation of this macro function. To send a message to an object, the *Send* function simply accesses the first field in the data structure of the object. To accomplish method inheritance, if the message-handler of a class does not recognize a message, it forwards the message to its base class. Thus, the method corresponding to a given object and message pair is dynamically searched for in the class inheritance tree upward from the object which receives the message.

**An Excerpt from OOPC.h**

```
typedef int (*Functionp) ();
typedef struct {
    Functionp dispatch;
} *Object;
#define Send(obj, msg, param, result) ((*obj->dispatch))(obj, msg, param, result)
```

Figure 4.3 Message-passing function *Send* (from Linowes, 1988)

Linowes's approach seems impractical for implementation of object systems which have many inheritance levels. Too much work will be involved in searching for a method. Adding inheritance levels to a system will make it slower. Inheritance is one of the main object-oriented techniques for reusability. Although some overhead is inevitable, a situation of direct conflict between reusability and efficiency is not acceptable (Meyer, 1988).

#### 4.2.2.2 Use of Self-Sufficient Objects

There is another approach which has been suggested by Bailey (1989) and discussed by Meyer (1988). An object can be viewed as carrying along at run-time the operation functions that are applicable to it. This view leads to an approach in which the pointers to methods are included in the data structure of an object class. Objects defined in such a way are called "self-sufficient" (Meyer, 1988). Figure 4.4 shows the definition of a self-sufficient class *REAL\_STACK*. The first two fields in the structure are the properties, and the others are pointers to operation functions of this class. These function pointers should be initialized so that they will point to appropriate functions at the creation time of an instance of this structure. This approach may work well for small scale applications which involve only a small number of objects. However, it implies that every instance of every class physically contains references to all functions that may be applied to it. Thus, this approach is prohibitive for large scale applications because of the space overhead.

```
typedef struct {
    int last;
    float impl[MAXSIZE];
    void (*pop)();
    void (*push)();
    void (*top)();
    BOOL (*empty)();
} REAL_STACK;
```

Figure 4.4 The definition of a self-sufficient object class (from Meyer, 1988)

#### 4.2.2.3 Use of Method-Dispatch Tables

It should be noted that the methods of a class are common to all instances of the class. Another structure for a class can be defined which contains pointers to all the operation functions of that class. This structure is called the "class descriptor" or "method-dispatch table". Only one reference in the structure of a class is necessary for reference to the method-dispatch table. This idea is the basis for the implementation of object-oriented programming languages which use C as the target language (Meyer, 1988).

This approach is used in the implementation of GUIDES (Zhang et al., 1990b). The method-dispatch table of a class is implemented as an array of pointers to methods of that class. These methods are defined as *static*, and are located in the same source file as the method-dispatch table. Messages are predefined integer constants, and used as indices of the corresponding method in the array. Pointers to method-dispatch tables of every object class in GUIDES are stored in an array which is called the "class-dispatch table". Class identifiers, which are also predefined constants, are used as the indices of these method-dispatch tables in the array. The data structure of the *Basic* class contains a class identifier. Thus, a message-passing function can find the operation function efficiently for a given object and message pair according to the identifier carried by the object and the message.

Several message-passing functions are used in the implementation of GUIDES, and each message-passing function corresponds to a specific message. To accomplish method inheritance, a message-passing function makes calls to appropriate methods either in a forward order (it calls the methods in the base class first, and then the methods in the derived class), or in a backward order (it calls the methods in the derived class first, and then the methods in the base class). If a class does not implement a method corresponding to a message, only the methods in its ancestor classes will be executed. Figure 4.5 shows a GUIDES

```

/*
 * GSi_CreateAgent - create a guides agent
 * calling order: basic class, composite class, specific agent class
 */
Gs_Agent
GSI_CreateAgent(resources, reso_count)
Gs_ResoList  resources;
PosInt      reso_count;
{
    auto PosInt      reso_name;
    auto Pointer     reso_value;
    auto Gs_Agent    agent;
    auto Gs_AgentFunction  funct;

    /* calling the basic class first */
    if (nilAG == (agent = sbpfMainTable[GSA_BASIC][GSO_CREATE]
                                     (agent, resources, reso_count)))
        return nilAG;

    /* calling the composite class */
    if (IS_COMPOSITE_AGENT(agent)) {
        if (nilAG == sbpfMainTable[GSA_COMPOSITE][GSO_CREATE]
            (agent, resources, reso_count))
            return nilAG;
    }

    /* finally calling the specific agent class */
    funct = sbpfMainTable[agent->class][GSO_CREATE];
    return (nilAF == funct) ? nilAG : (*funct)(agent, resources, reso_count);
}

```

Figure 4.5 A Message passing function of GUIDES

message-passing function, *GSI-AgentCreate*, for the message CREATE.

The weakness of the implementation of message-passing in GUIDES is lack of flexibility:

1. It is hard to add another level to the inheritance hierarchy (currently there are three levels). This would involve changes in all message passing functions.
2. A class can not overwrite the methods defined in its ancestor classes. This limits the number of choices in design.
3. A class has to know every message to build its method dispatch table, even if a message has nothing to do with a class. This implies that any change in the definition of messages involves changes in the code of all classes.

These problems are solved in CLOOP.

#### 4.2.3 CLOOP: A General Utility for OOP in C

The purpose of CLOOP is to provide systematic support and guidance for object-oriented programming in C. CLOOP consists of 7 functions, and is quite small in terms of its number of lines-of-code. This section discusses briefly the basic ideas of CLOOP. A detailed explanation of this utility may be found in (Zhang et al., 1990a). This utility was developed in the late stages of the GUIDES development, and was used in the development of the PlotManager of GUIDES.

#### 4.2.3.1 Representation of Objects

Following common conventions in C programming, the implementation of an object class usually consists of two files, a *class.h* file which contains the definition of the structure type of the class, and a *class.c* file which contains the implementation of the methods of the class. The designation *class* stands for the name of the class. A class can only recognize the structure type of another class by including the *class.h* file of that class in its implementation.

To accomplish message-passing and property inheritance, an object is represented in CLOOP as a pointer to a data structure, *ObjectRec*, or as a variable of type *Object* as shown in Figure 4.6. The field *class* in the *ObjectRec* data structure is the class identifier of an object. Each class has a unique identifier. The field *base* is of type *Object* and refers to an object of the base class if the object is of a derived class. *property* is a pointer to the class structure type of the object (note that *Pointer* is the general pointer type as mentioned previously). In the methods of a certain class where the structure type is known, *property* can be casted as a pointer to the class' structure type to access the fields in the data structure.

```

An Excerpt from cloop.h

typedef struct _ObjectRec {
    int          class;          /* class identifier */
    struct _ObjectRec *base;     /* base class object */
    Pointer      property;      /* its own property */
} ObjectRec, *Object;

```

Figure 4.6 Data structure of a general object class in CLOOP

In the case of inheritance, an object of a derived class is physically represented by several objects in memory. Each of these objects corresponds to a class in the class hierarchy above the class of the object, and these objects are linked together by the pointer *base*. Figure 4.7 shows the memory organization of an object of class *C*, which is derived from class *B*. Class *B* is in turn derived from class *A*.

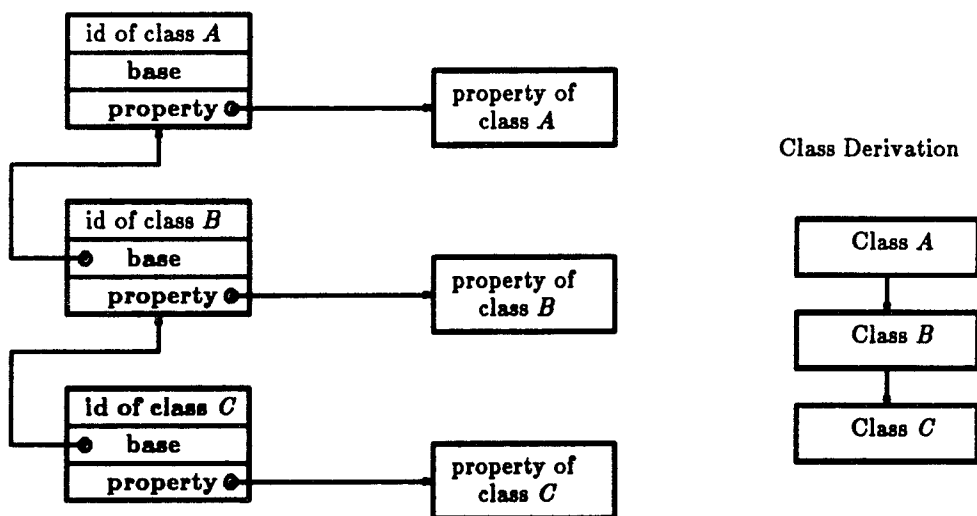


Figure 4.7 Memory organization of an object of a derived class, class *C*

It is clear that in this representation of objects, both encapsulation and property inheritance are accomplished without any conflict. The structure type of a class is hidden by default from other classes, including its derived classes. However, if it is desired, the *class.h* file of a class can be included in the implementation of its derived classes in order to access the properties of that class directly.

#### 4.2.3.2 Implementation of Message-Passing

In CLOOP, messages are represented by message identifiers which are positive integers. Identifiers of a set of messages for a class or for a group of classes are numbered continuously starting from zero.

Object classes which respond to the same set of messages are grouped as a sub-system. Classes in the same sub-system are usually in the same level of an abstraction hierarchy and respond to the same set of messages. For example, different element types in a finite element analysis program can be grouped in a sub-system. There are usually several sub-systems in an object-oriented application. CLOOP can handle more than one sub-system.

To accomplish polymorphism, a method of a class can only be invoked via the method-dispatch table of that class. Methods of a class are usually implemented as *static* functions in the *class.c* file. With CLOOP, a method should be defined as a function returning a *Pointer*. A general form of a method is shown in Figure 4.8, where *message* is the message identifier; *receiver* is the object which receives the message; and *param1* and *param2* are the parameters associated with the message. Because the two parameters are of type *Pointer*, any type of data can be carried along with them. The actual types of the parameters are determined uniquely by the associated *message*.

CLOOP holds the method-dispatch tables for every class. Functions are provided to build the method-dispatch table of a class. Each class needs to register itself as well as its methods to CLOOP. Also, it needs to implement a "method-registration function" for registering its methods to CLOOP. In this function, if the class is a derived one, the method-registration function of its base class is called first to register the methods implemented by the base class to its method-dispatch table. It then registers the derived class' methods to the table. Thus, a class can inherit the methods from its ancestor classes, or it can overwrite these methods by its own methods.



```

static Pointer
aMethod(receiver, message, param1, param2)
Object    object;
Message   message;
Pointer   param1, param2;
{
    (implementation of the method)
}

```

Figure 4.8 General form of a method

Figure 4.9 illustrates the method-dispatch table of a class *C* which is derived from a class *B* and which is in turn derived from a class *A*. The class *C* only implements 3 of the 8 methods which respond to the messages that it recognizes. It inherits the other 5 methods from its ancestor classes, class *B* and class *A*. The methods corresponding to the message 0 and 1 implemented by class *C* overwrite the methods corresponding to the same messages implemented by class *A* and *B* during the process of building the method-dispatch table.

CLOOP also holds the class-dispatch table for an application. The class-dispatch table is implemented as an array where each entry is a pointer to the method-dispatch table of a class. Identifiers of classes are used as indices of these pointers to method-dispatch tables. Functions are provided to build the class-dispatch table.

For a large scale application, the number of classes involved may be high. It is impractical to implement class identifiers as predefined integer constants, especially in a case where object classes are maintained in an object library. With CLOOP, class identifiers can be implemented as integer variables. CLOOP assigns an integer number as the first identifier for the classes in a sub-system when the sub-system is initialized. Figure 4.10 illustrates the memory organization of the class-dispatch table and method-dispatch tables for an

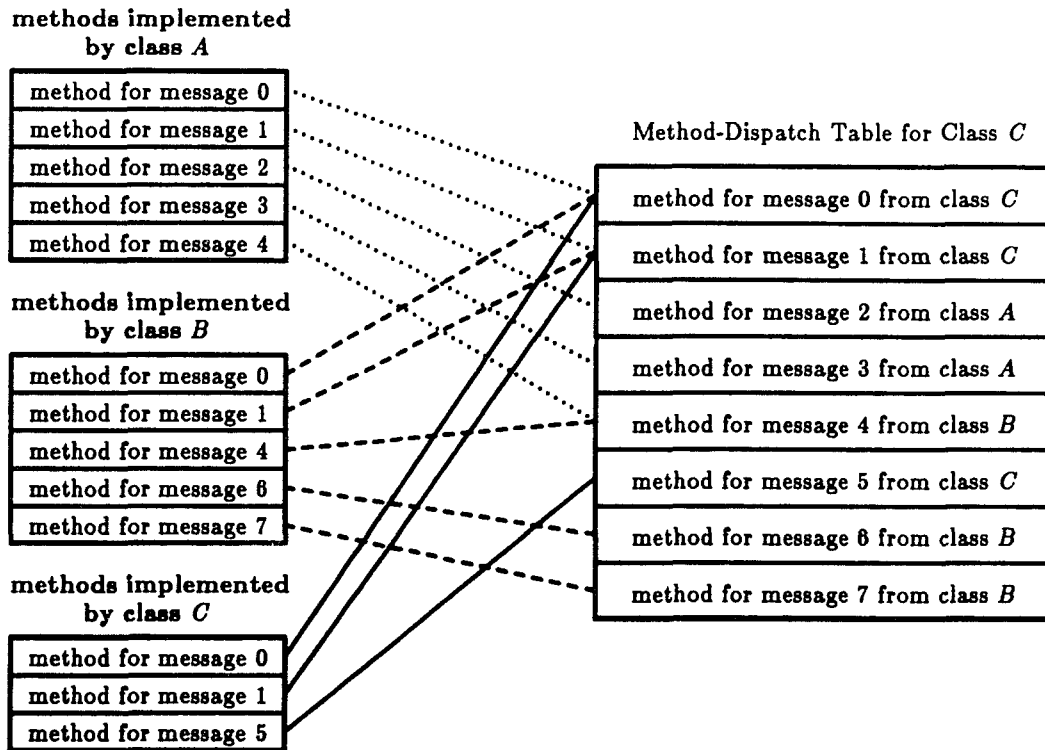


Figure 4.9 Method-dispatch table for a derived class, Class C

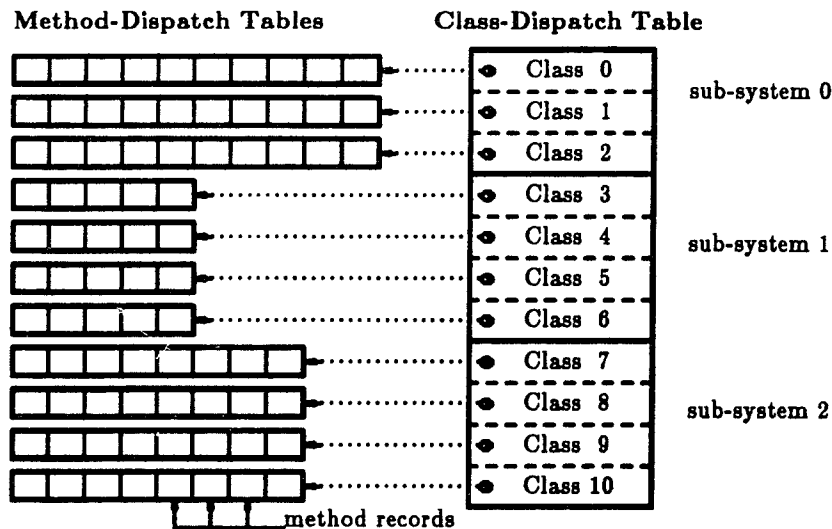


Figure 4.10 Class-dispatch table for an application

application.

As discussed previously, the efficiency of finding a pointer to an appropriate function for a given object and message pair is essential. In CLOOP, this task is performed by a function named *SendMessageTo*. *SendMessageTo* requires four parameters for the methods of any class, as described previously. An abstracted representation of *SendMessageTo* is shown in Figure 4.11. The name of this function is mnemonic in that the following call to the function

`SendMessageTo(curve, DRAW, ....)`

can be read as "send message to the curve with the message DRAW ....".

For a given *receiver* and *message* pair, *SendMessageTo* first obtains the method record from the class-dispatch table according to the class identifier carried by the *receiver* and the *message*. If the method record is empty, which

```

Pointer
SendMessageTo(receiver, message, param1, param2)
Object    receiver;
Message   message;
Pointer   param1;
Pointer   param2;
{
    auto ObjectMethod method;
    auto Object    object_tmp;
#ifdef DEBUG
    (error checking)
#endif
    if (nilOM == (method = ClassDispatchTable[receiver->class][message]))
        return nilP;
    object_tmp = receiver;
    if (object_tmp->class != method->class_id)
        object_tmp = FindBaseObject(object_tmp, method->class_id);
    return method->method_func(object_tmp, message, param1, param2);
}

```

Figure 4.11 An abstracted representation of *SendMessageTo*

means the class of the *receiver* does not have a method corresponding to the *message*, *SendMessageTo* will return null.

If the method record is not empty, *SendMessageTo* then checks if the method is implemented by the class of the *receiver*. If it is not, it assumes that the method is implemented by a base class of the *receiver*. It then calls the function *FindBaseObject* of CLOOP to find the base object for which the method is implemented. The method is finally called with the *receiver* or the base object as the first parameter, and the other three parameters are the same as the parameters passed to *SendMessageTo*.

This implementation is efficient because:

1. It finds whether the class of the receiver has a method corresponding to the message in one step;

2. If the method is implemented by a base class of the receiver, it needs at most  $m$  steps (where  $m$  is the number of ancestor classes of the receiver) to find the base object from the receiver by using the field *base* in the *ObjectRec* data structure (Figure 4.6). Each step consists of a structure selection operation and an integer equality operation.

With CLOOP, the implementation of an object-oriented system in C is more easily accomplished. In summary, the following aspects are the essential features of the CLOOP package:

1. The primary features of object-oriented programming are successfully supported by CLOOP; these include abstraction, encapsulation, polymorphism, and inheritance.
2. Although the level of class inheritance is not limited, only single-path inheritance is supported at present by CLOOP. That is, classes are organized in inheritance trees instead of inheritance graphs. With minor modification, support for multiple-inheritance can also be achieved.
3. The memory overhead is 8 bytes per object in the current implementation of CLOOP. This is the same as in the implementation of the Eiffel language (Meyer, 1988, p.344). Thus, if memory space is a problem for an application, the use of a large number of small objects should be avoided. Memory overhead due to the method- and class-dispatch tables is usually negligible.
4. CLOOP implements dynamic binding by using the class- and method-dispatch tables. Therefore, the method invoked by sending a message to a dynamically created object can only be determined at run time.

### 4.3 Programming in the C++ Language

The C++ language is a superset of the C language. It improves on C through its systematic support for object-oriented programming. It is a strong-typed language, and thus also improves on C through its type safety. It compensates for C's weaknesses without compromising C's strengths. The C++ language is distinguishable from other object-oriented languages such as Smalltalk by a variety of factors. Among these factors are: (1) emphasis on program structure, (2) flexibility of encapsulation mechanisms, (3) run-time efficiency, and (4) portability (Stroustrup, 1988b). Currently it is being used for large-scale software development in companies such as Apple, Apollo, AT&T, and Sun. It has been applied to many branches of programming, including CAD, database management, graphics, and scientific programming. The support of object-oriented programming by the C++ language is summarized in this section. Use of C++ in the present work is described in PART THREE of this thesis.

#### 4.3.1 Support of Abstraction and Encapsulation

The properties of a class are called member variables, and the methods are called member functions in C++. One goal of the C++ language is to make the manipulation of variables (objects) of user-defined abstract types as convenient and efficient as possible. To this end, methods can be defined for a class to properly create, initialize, and destroy its objects. Also, methods of a class can also be defined to convert objects of other classes to objects of that class. This is called user-defined coercion. Furthermore, operators can be redefined for a class such that objects of that class can be operated on by these operators directly in coding, e.g., such that a matrix multiplication can be written as  $C = A * B$ . This is called operator overloading.

Function names (including names of member functions) can also be overloaded, i.e., more than one function may share a single name in the same scope. These functions are distinguished from each other by their parameter lists which are referred to as their signatures. When the function name is used, the correct function is selected from these functions according to the actual parameters passed to the function and/or the type of the object through which the function is invoked. This feature makes the development of standard interfaces among different classes possible.

Manipulations on an object of a certain class usually can only be done through calls to the member or friend functions which may contain only a few lines of code. To accomplish encapsulation without sacrificing run-time efficiency, C++ allows a function (which can be member or friend function) to be declared as *inline*. Each call of an inline function is replaced by the copy of the entire function, much like macro function substitution by the C preprocessor. This mechanism eliminates the overhead of calling a function, and makes encapsulation practical.

The feature of parameterized types or parameterized classes is also supported by the C++ language even though this feature is not supported as well as it should be in the current release of C++ (release 2.0). A parameterized class is especially useful for the implementation of general data structures such as arrays, linked-lists, stacks, and queues. A parameterized class is implemented independently of the actual data type of the contents stored by the data structure. For example, a parameterized array class implements operations common to arrays of any element type. The parameterized class can be used to create arrays of integers, arrays of floats, or arrays of any user-defined type.

Finally, in the C++ language, the external specification of a class is naturally separated from the implementation. The declaration of a class is its external specification, and it is usually provided in the header file of the class by

C++ convention. The declaration of a class contains the declaration of member variables of the class and the prototyping of member functions. The prototype of a function is a template of the function which the C++ compiler uses to ensure that proper arguments are passed when the function is called. The information contained in the class declaration is all that the clients of the class need to know to communicate with or use objects of the class. The header file of a class must be included by the code of its clients such that the C++ compiler will be able to detect any incorrect calls to functions of that class.

#### 4.3.2 Support of Inheritance and Polymorphism

The C++ language allows hierarchically organized classes to be defined to represent a hierarchy of abstractions. The inheritance features of OOP are fully supported in C++. Polymorphism is supported partly by function name overloading and operator overloading, and partly and more importantly by the mechanism of virtual functions. A member function in a base class can be declared as a virtual function. A virtual function is one that can be overridden by a derived class's member function which has the exact same type and same parameter list as specified in the virtual function. This mechanism is important in the implementation of class inheritance. An object of a derived class can be treated as an object of its base class in the C++ code. However, invoking a virtual function through an object of a derived class actually results in the function defined in the derived class being called even if the object is treated as an object of the base class.

The C++ language also supports abstract classes. An abstract class is one that is designed to provide the definitions of properties and methods for its derived classes to inherit, and is not used to create objects. It provides a framework and specifies a standard interface for its derived classes. This mechanism provides an important tool for creating extensible systems.



### 4.3.3 Support of Composition

Composition of objects is also supported in the C++ language. A member variable of a class can be an object of another class. Such a member variable is called a member object. Instructions are provided in C++ to create and initialize a member object when an object is created, and to destroy the member object when the object is destroyed.

## 4.4 OOP for Engineering Software Development

Object-oriented programming in an object-oriented language facilitates a new style of software development: software development based on a large number of prefabricated reusable software components (or object classes). Object-oriented programming and object-oriented languages do not make software reuse happen, but they make it feasible and practical. Substantial applications can be built efficiently based on reusable software components accumulated from previous software projects. This new style of software development should be more productive than previous styles. Software developed in this way will be less error-prone, more abstract, more readily modified, and more extendible.

However, it may be questioned by some that the full advantage of an object-oriented language such as C++ can only be realized by mastering the techniques of object-oriented programming. Object-oriented programming at the present time is unfamiliar even to many computer science professionals, while many scientific and engineering software developers are even less educated in object-oriented techniques. This is indeed a major obstacle to the adoption of this new style of software development in engineering computing. However, to create good object classes for a certain area in engineering computing, knowledge in both the application-specific domain and in object-oriented programming techniques are necessary. Without a good-understanding of any

one of these two aspects, the abstraction hierarchy for the application-specific domain can not be properly defined, nor can these abstractions be implemented into a set of quality object classes.

However, in many aspects this new style of software development can be performed more easily than the traditional styles of software development. This is because object-oriented programming makes a clearer distinction between the developers of reusable components (object classes) and the users of these components.

From the viewpoint of users of object classes, an object class is just a data type that programmers can use in the same way as built-in types of the language. No internal details of these types need to be known to use them. This is similar to the aspect that one does not have to know how an integer variable is represented and operated on internally to use an integer variable. A variable of a certain class can be created as easily as variables of built-in types. The destruction of such variables also will be taken care of by the compiler once they are out of scope. They can be operated on easily by following their class specifications, especially when operators are overloaded for their classes. Thus, programming in an object-oriented language by use of existing object classes does not necessarily require a sophisticated training in object-oriented programming. If a large collection of object classes are provided, it may be easier to perform than programming in traditional languages for many software projects.

Some may also be concerned that the run-time efficiency of object-oriented languages is not sufficient for numerical computation. Object-oriented programming performed in an object-oriented language does add overhead at run-time. However, it should be noted that not all engineering software is computationally intensive. In typical engineering software, often the portion which is computational intensive is relatively small in size, and the bulk of the

program text is related to manipulation of data structures, storage management, input and output, pre- and post-processing, etc. In most cases, this portion grows much faster than the purely numerical part especially with the increasing demand for interactive computer graphics and graphical user-interfaces.

Moreover, due to the rapid increase of computer processing speed, the efficiency of a certain language for numerical computation has become less important now than ever before. This trend is also expected to be continued. Thus, efficiency needs to be a primary concern only for software component involving large scale computation.

To obtain a rough estimate of the run-time efficiency of the FORTRAN, C, and C++ languages for numerical computation, benchmark tests for the matrix classes, which will be described in PART THREE, have been performed (Zhang, 1990c). In these tests, programs were written in the FORTRAN and C languages in a conventional format, and in the C++ language using the matrix classes to evaluate certain matrix expressions for a large number of times. The test results showed that the C implementation was the most efficient one, and that the FORTRAN implementation was slightly slower. The C++ program used 20% to 50% more CPU time than the C program for different matrix expressions. The author believes that this is a reasonable trade-off for the benefits obtained from object-oriented programming and the C++ language.

## CHAPTER 5 SPECIFIC ISSUES RELATED TO THE SESDE DEVELOPMENT

The general software engineering principles and technologies related to the development of the SESDE have been reviewed in the preceding chapters. Specific considerations in the application of these principles and technologies in the current research are discussed in this chapter. These considerations include the design of reusable components for the SESDE, and the development of applications in the SESDE.

### 5.1 Design of Reusable Components

Identifying and defining individual reusable components as well as establishing the class inheritance hierarchy for a specific problem domain is not a trivial or an automatic process. It is a process independent of any specific programming language. Conceptual clarity, runtime efficiency, ease of development and use, and ease of future extension all need be considered in achieving the goal of reusability. Some considerations in the design of reusable components for the SESDE are described in the following sections.

#### 5.1.1 Object-Oriented versus Functional Design

As described in Section 3.3, object-oriented design is superior to functional design in many aspects for achieving reusability. Thus, the object-oriented design approach is adopted in the development of SESDE. However, this does

not exclude use of the functional approach in the development of software components. For some conceptually simple and distinct operations or computations specified by a small set of parameters, the functional approach can also lead to good component design.

As an illustration example, Figure 5.1 shows an abstract form of a reusable component, *Root*. This component is used for searching a root of a nonlinear equation,  $f(x) = 0$ , in a given range of  $x$  based on a functional design. An example application using the component is also shown in the figure. There are two functions in the component: the function *SearchRoot* provides the major service of the component, i.e., it searches for a root of the given equation in a given region; the other function, *SetRootPrecision* is used to change the default value of convergence tolerance used in searching for the root. If the required operation on a nonlinear equation in an application is only to find the roots of an equation, this functional design leads to a simple and clean client interface of the component.

On the other hand, if some other operations on a nonlinear equation are required in an application, e.g., performing numerical differentiation and numerical integration, an object-oriented design may be better than the functional design for this component. Figure 5.2 shows the specification of the component written in the C++ language in an object-oriented design as well as the same example application. As seen from Fig. 5.2, if the application only searches a root of a nonlinear equation, the component implemented with an object-oriented design is not necessarily easier for the application than implemented by the functional design. Moreover, the component designed in the functional approach may be easier to use in an application written in a language other than C++, such as FORTRAN.

Thus, the development of reusable components should not be restricted only to object-oriented design. Both object-oriented and functional design

**A functional design of the Root component**

```

static double precision = 0.001;
double SearchRoot(equation, x1, x2, error)
double (*equation)(); /* pointer to the function implementing a nonlinear equation */
double x1, x2; /* specify the region of x */
int error; /* error flag */
{
    implementation details
}

void SetRootPrecision(prec)
double prec; /* precision wanted by a client */
{
    precision = prec;
}

```

**An application using the Root component**

```

double nonlinear_equation(x)
double x;
{
    computing the value of the nonlinear equation at a given x
}

int main()
{
    double x, a, b;
    int error;

    printf(" Enter the definition of the region (a, b):");
    scanf(" %lf %lf", &a, &b);
    x = SearchRoot(nonlinear_equation, a, b, &error);
    if (0 == error)
        printf("The root is %f", x);
    else
        printf("Can not find root");
    exit (0);
}

```

Figure 5.1 Functional design of the *Root* component

### An object-oriented design of the Root Component

```
class Equation {
    (*double) () equation;
    double precision;
public:
    Equation(const (*double)()); /* constructor */
    double SearchRoot(double a, double b, int *error); /* method for searching a root */
    double Integration(double a, double b, int *error); /* method for integration */
    double Differentiation(double a, int *error); /* method for differentiation */

    other methods
};
```

### An application using the Root component

```
double nonlinear_equation(x)
double x;
{
    computing the value of the nonlinear equation at a given x
}

int main()
{
    Equation eq(nonlinear_equation);
    double x, a, b;
    int error;

    printf(" Enter the definition of the region (a, b):");
    scanf("%lf %lf", &a, &b);
    x = eq.SerachRoot(a, b, &error);
    if (0 == error)
        printf("The root is %f", x);
    else
        printf("Can not find root");
    exit(0);
}
```

Figure 5.2 Object-oriented design of the *Root* component

methodologies need to be considered in the development of components similar to that of the *Root* component described above. The decision on the design method depends on the conceptual complexity of the component and the requirements of the component's clients.

### 5.1.2 Inheritance versus Composition

There are many different ways to construct objects for a specific problem domain. As described in Section 4.1, objects or classes of objects are abstractions of real world entities. A class of complex objects may be represented by a series of abstractions, each of which is represented by a class distributed in a class inheritance hierarchy. Each of these classes maintains the properties to express its corresponding abstraction, and each implements the methods necessary to the abstraction. Thus, code duplication is avoided because similar object classes can be derived from the same series of base classes.

More importantly, complicated operations for a class of complex objects can be decomposed into a series of relatively simple operations performed by its base classes as well as the class self. Also as described in Section 4.1.5, object composition is an alternative mechanism to avoid code duplication and reduce code complexity. A powerful object-oriented language, such as the C++, provides support for both mechanisms.

Thus, when an existing class is similar in some aspects to the class being defined, questions may arise that whether the new class should be defined: (1) as a derived class from the existing one, or (2) as a sibling class derived from the same base class as the existing one, or (3) as a class whose instance is composed of an instance of the existing one. The class being defined is put in a lower abstraction level than the existing one by the first approach, in the same abstraction level as the existing one by the second approach, and in a branch of the class inheritance hierarchy tree (which may be different than the existing



one) by the third approach.

As an example, consider the following relationship between two classes: one implements a 4-node, two-dimensional, solid element type, and the other implements an 8-node, two-dimensional, solid element type for finite element analysis. The class of the 4-node element type is derived from a class implementing a general two-dimensional solid element type. It implements properties for expressing the shape functions of 4-node elements, and methods manipulating these shape functions including evaluating function values and function derivative values at a given point.

From the above discussion, there are three ways to define the class of the 8-node element type:

1. The class of the 8-node element type is defined as a derived class of the class of the 4-node type. In this case, it implements properties expressing the shape functions of middle nodes and methods for manipulating these shape functions.
2. The class is defined as a derived class of the class of the general element type, and a sibling of the class of the 4-node type. In this case, it implements properties and methods for the shape functions of all the nodes of a 8-node element.
3. The class is defined such that its instance is composed of an instance of the class of the 4-node type. In this case, it contains a pointer to an instance of a 4-node element and implements properties and methods for the shape functions of middle nodes.

By the first and third approaches, code duplication for manipulating the shape functions of the corner-nodes can be avoided in the implementation of the class of the 8-node element type. However, because the 4-node and 8-node element types conceptually are of the same level of abstraction (that is, they are

all special cases of the general element type), the second approach is more logical than the others. Even though in the second approach the code in the class of 4-node can not be reused for the 8-node class, the code duplication is not significant. Moreover, the code complexity resulting from the second approach may be lesser than that of the other two approaches.

In fact, there is another approach usually adopted in finite element programming: taking the 4-node element type as a special case of the 8-node element type and implementing the 8-node element type only. This is a common approach in which simple cases are reduced from a general and complex one. However, the complexity of the resulting code by this approach is often high, and should therefore be avoided.

In short, the aspect of where and how to use inheritance and composition mechanisms in defining an object system for a specific problem domain needs careful investigation. There are tradeoffs between conceptual clarity, code complexity, and code duplication.

### 5.1.3 Subsystems

A large object-oriented system tends to be composed of several levels of abstraction. Classes in the same level are often derived from the same base class. The classes of two-dimensional solid element types discussed in the previous section can be designed in the same level. Classes in the same level should have a standard interface to their clients (i.e., they should accept the same set of messages, and accept the same parameter list associated with each message). Only by a standard interface, can these classes be treated equally by their clients, and therefore be replaced or added to an application without affecting their clients.

Several levels of classes related by an inheritance mechanism can be further grouped together forming a subsystem implementing a higher level abstraction. A subsystem may be called a software tool. GUIDES is such a subsystem designed for creating and handling the graphical user-interface of applications. A subsystem can be a "white-box" or a "black-box" to its clients. The clients of a white-box subsystem can, and have to, communicate with objects in the subsystem directly. Clients can also derive their own classes from the classes in the subsystem. The objects and the internal structure of a black-box subsystem are invisible to its clients. This type of subsystem is enveloped by a set of conventional interface functions. Clients can only communicate with the objects through these functions.

The benefits and disadvantages of these two approaches are clear. In the white-box approach, clients can communicate with the objects in the subsystem directly. This may lead to a higher runtime efficiency. Clients can also derive their own classes from the classes of the subsystem. This leads to a higher code reusability and extendibility. However, the clients of a white-box subsystem are forced to follow the same design methodology as the subsystem (i.e., object-oriented design), and are forced to use the same implementation language as the subsystem (such as C++). Moreover, the clients may tie their own class hierarchy together with the subsystem, thus leading to a complicated program structure.

The disadvantages of the white-box approach are the benefits of the black-box approach. A client can be designed using whatever methodology and implementation language that are appropriate to it. More importantly, if clients are designed in an object-oriented manner, the class hierarchy of clients is separated from the class hierarchy of the subsystem. This leads to a conceptually clean program structure.

Whether a white-box approach or a black-box approach should be used for a subsystem depends mainly on the necessity for the clients to derive their classes from the classes in the subsystem. There are also tradeoffs between flexibility and conceptual clarity. GUIDES is developed using a black-box approach. The GUIDES development will be discussed further in PART TWO.

## 5.2 Applications Development

From the discussions in Section 3.4, both the operational model and the knowledge-based model are appropriate for the development of applications in an environment such as the envisioned SESDE consisting of a large collection of reusable software components and software tools. The knowledge-based model is superior to the operational model in the aspect of software productivity. However, it needs extensive programming tools which may not be available in the near future.

Thus, the development of applications in the early stage of the SESDE should follow the operational model. A prototype of an application would be built first according to its specification by using the existing reusable components and software tools. Feedbacks can then be obtained from both the end-user and the designer. The prototype would be then modified based on these feedbacks. The process would be repeated until both the end-user and the designers are satisfied. The last stage in this iterative development is to refine and optimize the prototype to build the final product of the application.

## LIST OF REFERENCES

1. Bailey, S. C., "Designing with Objects", Computer Language, January, 1989, pp. 34-43.
2. Barbacci, M. R., Habermann, N. A., and Shaw, M., "The Software Engineering Institute: Bridging Practice and Potential", IEEE Software, November, 1985, pp.4-21.
3. Bassett, P. G., "Frame-Based Software Engineering", IEEE Software, July 1987, pp.9-16.
4. Bishop, P., "Fifth Generation Computers", Ellis Horwood Limited, 1986, 166pp.
5. Burton, B. A., Aragon, R. W., Bailey, S. A., Koehler, K. D., and Mayes, L.A., "The Reusable Software Library", IEEE Software, July 1987, pp.25-33.
6. Coad, P., and Yourdon, E., "Object-Oriented Analysis", Prentice-Hall, Inc., 1990.
7. Cox, B., and Hunt, B., "Objects, Icons, and Software-ICs", in Tutorial: Object-Oriented Computing, Vol.2, Implementations, Edited by Gerald E. Peterson, Computer Society Press, 1987, pp.99-108.
8. Cox, B., "The Object-C Environment", in Digest of Papers, CompCon88, Thirty-Third IEEE Computer Society International Conference, San Francisco, CA., Feb. 29 - Mar. 4, 1988, Computer Society Press, pp.166-169.
9. Depledge, P. G., "Software Engineering Terms and Concepts", in Software Engineering for Microprocessor Systems, edited by P.G. Depledge, 1984, Peter Peregrinus Ltd, 261pp.
10. Ellison, R., "Trends in Software Development Environment for Large Software Systems", in Digest of Papers, CompCon88, Thirty-Third IEEE Computer Society International Conference, San Francisco, CA., Feb. 29 - Mar. 4, 1988, Computer Society Press, pp.259-261.
11. Fischer, G., "Cognitive View of Reuse and Redesign", IEEE Software, July 1987, pp.60-72.

12. Ingraffea, T. and Mink, K., "Project SOCRATES: Fostering A New Collegiality", *Academic Computing*, October 1988, pp.20-21, 60-63.
13. Kaiser, G. E. and Garlan, D., "Melding Software Systems from Reusable Building Blocks", *IEEE Software*, July 1987, pp.17-24.
14. Kernighan, B. W., and Ritchie, D. M., "The State of C", *BYTE*, August 1988, pp.205-210.
15. Lenz, M., Schmid, H. A., and Wolf, P. F., "Software Reuse through Building Blocks", *IEEE Software*, July 1987, pp.34-42.
16. Linowes, J. S., "It's an Attitude", *BYTE*, August 1988, pp.219-224.
17. Love, T., "The Economics of Reuse", in *Digest of Papers, CompCon88, Thirty-Third IEEE Computer Society International Conference*, San Francisco, CA., Feb. 29 - Mar. 4, 1988, Computer Society Press, pp.238-241.
18. Meyer, B., "Software Engineering for Engineering Software", in *Tools, Methods and Languages for Scientific and Engineering Computation*, Edited by B. Ford, J. C. Rault, F. Thomasset, Elsevier Science Publishers B. V. (North-Holland), 1984.
19. Meyer, B., "Object-oriented Software Construction", 1988, Prentice Hall, 543pp.
20. Meyer, B., "Reusability: The Case for Object-Oriented Design", *IEEE Software*, March, 1987, pp.50-64.
21. Open Software Foundation, "OSF/Motif Programmer's Guide", Prentice Hall, New Jersey, 1990.
22. Pinson, L., and Wiener, R., "An Introduction to Object-Oriented Programming and Smalltalk", Addison-Wesley, 1988, 502pp.
23. Prieto-Diaz, R. and Freeman, P., "Classifying Software for Reusability", *IEEE Software*, January, 1987, pp.6-16.
24. Pyster, A. and Barnes, B., "The Software Productivity Consortium Reuse Program", in *Digest of Papers, CompCon88, Thirty-Third IEEE Computer Society International Conference*, San Francisco, CA., Feb. 29 - Mar. 4, 1988, Computer Society Press, pp.242-247.
25. Rappaport, A. and Hinder, V., "Market Impact of Integrated Software Development Environment", in *Digest of Papers, CompCon88, Thirty-Third IEEE Computer Society International Conference*, San Francisco, CA., Feb. 29 - Mar. 4, 1988, Computer Society Press, pp.250-253.
26. Shaw, M., "Abstraction Techniques in Modern Programming Languages", in *Tutorial: Object-Oriented Computing, Vol.2, Implementations*, Edited by Gerald E. Peterson, Computer Society Press, 1987, pp.146-161.

27. Sommerville, I., "Why Software Engineering", in Software Engineering for Microprocessor Systems, edited by P.G. Depledge, 1984, Peter Peregrinus Ltd.
28. Sommerville, I., "Software Engineering", Second Edition, 1985, Addison-Wesley, 334pp.
29. Stroustrup, B., "The C++ Programming Language", 1987, Addison-Wesley, Massachusetts, 328pp.
30. Stroustrup, B., "What is Object-Oriented Programming?", IEEE Software, May, 1988a, pp.10-20.
31. Stroustrup, B., "A Better C?", BYTE, August 1988b, pp.215-216D.
32. Thomas, D., "What's in an Object", BYTE, March 1989, pp.231-240.
33. Tracz, W., "Reusability Comes of Age", IEEE Software, July 1987, pp.6-8.
34. Wegner, P., "Capital-Intensive Software Technology", IEEE Software, July, 1984, pp.7-45.
35. Wegner, P. "Learning the Language", BYTE, March 1989, pp.245-253.
36. Wiegand, G., "HOOPS Reference Manual", Second Edition, Ithaca Software, 1988.
37. Zhang, H., White, D. W., and Chen, W. F., "A Library for Object-Oriented Programming in C", Structural Engineering Report, CE-STR-90-9, School of Civil Engineering, Purdue University, West Lafayette, Indiana, April, 1990a, 55pp.
38. Zhang, H., Tan, L., White, D. W., and Chen, W. F., "Design and Implementation of GUIDES: A Graphical User-Interface Development System", Structural Engineering Report, CE-STR-90-7, School of Civil Engineering, Purdue University, West Lafayette, Indiana, June, 1990b, 112pp.
39. Zhang, H., White, D. W., and Chen, W. F., "An Object-Oriented Matrix Manipulating Library", Structural Engineering Report, CE-STR-90-10, School of Civil Engineering, Purdue University, West Lafayette, Indiana, June 1990c, 77pp.

PART TWO

A GRAPHICAL USER-INTERFACE DEVELOPMENT SYSTEM



## CHAPTER 6 GRAPHICAL USER INTERFACE TOOLS

The intent of this chapter is to give a brief review of the current technology of Graphical User-Interface (GUI) tools. Recent achievements in the development of graphical user-interfaces are summarized in Section 6.1. The necessary separation of user-interface components from application-specific components and the benefits associated with the use of general graphical interface tools are described in Section 6.2. The architecture and characteristics of graphical user-interface development systems are presented in Section 6.3. Several issues that need to be considered in design of user-interface tools are discussed in Section 6.4.

Three different types of GUI tools are described in Sections 6.5, 6.6, and 6.7: the Macintosh GUIs (Mednieks et al., 1986; Schmucker, 1987; Shell, 1988), the X11 Toolkit (McCormack et al., 1988a, 1988b; Swick et al., 1988), and the GUI tools of the GRAFIC/CE88 system (Zhang et al., 1988a). The purpose of these sections is to illustrate the state-of-the-art of graphical user-interface tools in 1989. The design of GUIDES, the Graphical User-Interface DEvelopment System, has evolved from the study of the first two systems and from experiences in the development of GRAFIC/CE88. The OSF/Motif, a commercial graphical user-interface development system which has appeared on the market in 1990, is described in Section 6.8 to provide more updated information.

## 6.1 Introduction

In the last decade, engineering software has rapidly progressed from non-interactive to highly interactive programs. The common view of how programs should interface with the user has now changed to include the direct involvement of the user during execution. Since Apple introduced its revolutionary Macintosh computer in 1984, software interfaces have further progressed from being text-oriented to graphics-oriented. Interactive graphical user-interfaces have been referred to by many as the wave of the future (Seymour, 1989; Fiedler, 1989). They are an important feature of today's successful software systems, and they will be a necessary aspect of future software systems. At present, the graphical user-interface technology is still in its infancy. In fact, graphical user-interfaces are one of the major areas in software engineering research (Dodani et al., 1989; Fiedler, 1989; Greenberg, 1989; Hartson, 1989; Hayes et al., 1989; Hurley et al., 1989; Myers, 1989; Thompson, 1989).

The increased utilization of GUIs has been driven by the introduction of powerful workstations with bitmapped screens and pointing devices. Since pictures are easier to understand than text, software that has a GUI is easier to learn and use. Moreover, in many cases GUIs make the software more powerful. GUIs provide engineers and scientists the means to manipulate data directly in a graphical mode. They offer impressive advantages including better analysis, improved productivity, and increased cost savings.

At present, many operating and windowing systems provide support for the development of GUIs for application-specific software. The operating system of the Macintosh computer was the first system of this sort. This has given Apple a competitive edge in the personal computer market. MIT, in cooperation with DEC and IBM, has developed a standard graphics-based software platform called the X Window System. This system is well written, and serves as a base for the development of a variety of GUIs.

Another popular package is NeWS (the Network Extensible Window System) from Sun Microsystems. NeWS is based on the PostScript language. The Open Software Foundation (OSF) has finally released their GUI product, OSF/Motif, in 1990. This package is also based on the X Window System (Paul, 1989). NeXT's NextStep is a similar system for the NeXT computer. Comprehensive reviews on the state-of-the-art of GUIs can be found in (Fiedler, 1989; Greenberg, 1989; Hayes et al., 1989; Myers, 1989; Seymour, 1989).

Systems which support GUI development must provide an Application Program Interface (API) by which applications build their GUIs. An API is built on top of a windowing and/or graphics system depending on the architecture of the system. A windowing system is a set of programming tools for building interface components and for collecting user input and passing the input to applications running within the system. A graphics system defines how graphics are actually displayed on the screen. Windowing and graphics systems often provide tools for application programmers to create graphical interfaces and integrate the interface with their applications. These systems will be discussed further in the following sections.

## 6.2 The Need for GUI Tools

Interactive user-interface software is often large, complex, and difficult to debug and modify. An application's interface can account for a significant fraction of the code. Furthermore, the creation of a good user-interface is difficult. There are no guidelines or techniques which guarantee that a program will be easy to learn or use. The only reliable way to generate a quality interface is to test prototypes and modify the design based on user's comments. This may be referred to as interactive design.

Many programs have been designed with the user-interface component and application-specific components combined. The problem with this approach is

that it is extremely difficult to maintain and modify such a monolithic system. Dodani et al. (1989) illustrates this problem as follows. When WordStar's user interface was redesigned to produce WordStar 2000, the effort was nearly equivalent to writing an entire new program. Such an enormous effort suggests that little or no code from the previous version could be reused. This was most likely due to fact that the old user-interface was intimately intertwined with the code that supported the word processing application.

An application should be seen as consisting of separate user-interface and application-specific components. The benefits of separating the interface component from application-specific components are obvious. The interface and application-specific components can be designed, developed, tested, and modified separately. The interface component can be altered or even replaced without affecting the code that defines the application. Moreover, the separation leads naturally to the development and utilization of general interface tools.

Utilization of user-interface tools provides two main advantages over direct coding of the user-interface with the application-specific code (Myers, 1989):

1. It results in better interfaces:
  - It encourages interactive design. An interface can be rapidly prototyped and implemented, possibly even before the application-specific code is written.
  - Different applications will have more consistent interfaces because they have been created with the same user-interface tools.
  - It is easier to involve nonprogrammers such as graphic artists, cognitive psychologists, human factors specialists, and end-users in designing the interface.

2. The user-interface code is easier and more economical to create and maintain:

- It is better designed and has a higher reliability because it has been separated from the specific applications.
- It is more reusable because the user-interface tools incorporate common parts of many applications.
- It can serve as a crucial layer between applications and evolving user-interface and programming environment. Thus, it is easier to port an application to different environments, and to avoid early obsolescence of applications due to changes in the environment.

Due to these advantages, many tools have been created in the software industry to facilitate user-interface development. To build quality user-interface tools, two main principles need to be followed: reusability and encapsulation. User-interface tools should be designed as general tools so that they can be reused readily in diverse applications. The details about the implementation of a tool should be encapsulated such that the tool can be used without a detailed understanding of the underlying implementation. The object-oriented programming methodology supports these principles, and it has been widely used in the development of many existing user-interface tools. The classification of existing tools is considered next.

### 6.3 Current State of GUI Development Systems

Myers (1989) classifies user-interface tools in two general forms: user-interface toolkits and user-interface development systems.

### 6.3.1 User-Interface Toolkits

A user-interface toolkit is a library of interaction techniques. An interaction technique is a graphical tool or a way of using a physical input device (such as mouse or keyboard) to input a value (such as typing a key or clicking a mouse button) along with the feedback that appears on the screen. Typical interaction techniques are menus, scroll bars, and buttons operated with a mouse. Most windowing systems come with a toolkit that a programmer can use by writing code to invoke and organize the interaction techniques.

There are two kinds of toolkits. The most conventional one is a collection of procedures that can be called by application programs, such as the Macintosh Toolbox (Mednieks et al., 1986). The other kind uses the inheritance feature of the object-oriented programming paradigm. This makes it easier for the designer to customize the interaction techniques. The X11 Toolkit (McCormack et al., 1988a, 1988b; Swick et al., 1988) is an example of this kind of package.

The disadvantage of using toolkits is that they are often time consuming and difficult to use. A toolkit typically includes hundreds of procedures that implement different interaction techniques. It is often not clear how to use the procedures to create a desired interface. Thus, toolkits do not provide much support for the design of interfaces.

### 6.3.2 User-Interface Development Systems

The problems associated with toolkits have led to the creation of graphical user-interface development systems, such as the Apple MacApp. Apple developed this system after it found that people were having difficulty using the Macintosh Toolbox (Myers, 1989). A user-interface development system is an integrated set of graphical interface tools which can help programmers: (1) to specify the interaction techniques of an interface during the design phase, and (2) to create interaction techniques and manage the interaction between the

end-user and the application during run-time. A comprehensive development system should handle all aspects of an interface. This includes management of all portions of the display and all aspects of the interaction between the user and the application.

A user-interface development system is usually built on top of either a graphics or a windowing system. The architecture of the existing user-interface development systems varies. A typical development system may contain the following three components:

1. An interaction control component that handles event queuing and sequencing.
2. A programming framework that helps in constructing the interface component's interaction techniques and in establishing the connection between the interaction techniques and the application's semantics.
3. A user-interface construction set that enables the interactive construction of user interfaces without requiring an understanding of implementation details of the interface.

The details of each of these components is discussed below.

#### 6.3.2.1 Interaction Control Component

The interaction control component (or event manager) manages the communications between the end-user and the application. It collects the raw user input (i.e., the events) via the windowing or graphics system, interprets the events, and communicates the interpreted events to the appropriate code unit in the interface or the application-specific component.

### 6.3.2.2 Programming Framework

The programming framework provides facilities with which to: (1) construct the interface component of an application, and (2) manage the communications between the interface component and application-specific components. The framework is concerned with the presentation of application-specific information to the user, accepting user input via the event manager, and invoking application functions to perform the functionality of the application. In most development systems, frameworks are designed using an object-oriented approach. The effectiveness of a framework depends on the ease with which it can be used. Frameworks can be classified as either "white-box" or "black-box" (Dodani et al., 1989).

The use of a white-box framework requires knowledge of its internal data structures and implementation. A white-box framework provides a main driver for controlling and sequencing the activities and can be seen as an extensible skeleton. The interface techniques specified in the skeleton can be tailored to suit a particular application. Typical examples of the white-box approach are the MacApp framework (Schmucker, 1987) and the ICpak 201 framework (Cox et al., 1987).

The strength of a white-box approach is its flexibility for creation of different kinds of interaction techniques. A basic problem associated with the use of this approach is that it requires intimate knowledge of the underlying structure and implementation of the framework. This requires a steep learning curve for developers and makes the tools usable only by experts. A natural way to reduce the complexity of this approach is to provide a construction set that automates the process of building an interface.

In contrast, a black-box framework requires that a designer only provide a specification defining the interaction techniques for a specific application and the methods by which users interact with the application. Thus, a black-box



framework requires that the designer understand only the external interface of the major components of the framework. A special-purpose language, called the interface description language, is usually necessary to specify the interface. The framework reads and interprets the specification at run time to establish automatically the appearance of the interface, and to establish the connection between each individual interaction technique and between the interface and application-specific components.

The description language enables application programmers to define the characteristics of the user interface independent of the actual application code. Application programmers can make changes to the overall appearance and layout of an application without having to modify, recompile, or relink the application itself. A commercial example of a user-interface description language is the User Interface Language (UIL) for the OSF/Motif user-interface development system (Paul, 1989).

The main weakness of the black-box approach is that it usually provides only a limited number of interaction techniques. Also, it requires the interface designers to understand the description language. A construction set helps to alleviate this second weakness.

#### 6.3.2.3 Construction Set

A construction set is an interactive graphics tool for building the user-interface without requiring the user to understand the implementation details of the framework. It lets designers define the interface by: (1) selecting and placing the desired interaction tools of the framework on the screen, and then (2) describing interactively the connection between the interaction tools and between the tools and the application-specific code.

With a white-box framework, the construction set may produce the code which implements the interface. This code can be compiled and linked with the framework. With a black-box framework, the construction set may produce an interface specification in the description language of the development system. This specification can be used by the framework at run time to establish the interface.

A construction set provides the necessary basis for rapid prototyping and incremental development of user-interfaces. The philosophy behind the use of a construction set is that, because the visual presentation of the interface is one of its most important aspects, a graphical tool is the most appropriate way to specify the presentation. Construction sets are often very easy for the designer to use. Some of these systems can be used easily by non-programmers. Typical examples of construction sets are the ViewEdit for the MacApp framework (Dodani et al., 1989) and the Interface Builder (IB) for the NeXT computer (Thompson, 1989). The construction set itself is however very complicated to develop.

#### 6.4 Issues in Design of GUI Tools

The separation of the GUI from the applications raises three important questions in the design of interface tools: (1) how should the interface component be separated from applications such that both interface component independence and run-time efficiency can be satisfied; (2) how should the interface component and the application-specific components communicate at run-time; and (3) how should the user-application interaction process be controlled. These questions are addressed below.

### 6.4.1 Semantics of the Interface Component

Most user-interface tools are event-based. Figure 6.1 shows a simplified view of an application where the interface component is separated from the application-specific components. There are two types of dialogue which occur in this type of application: external and internal.

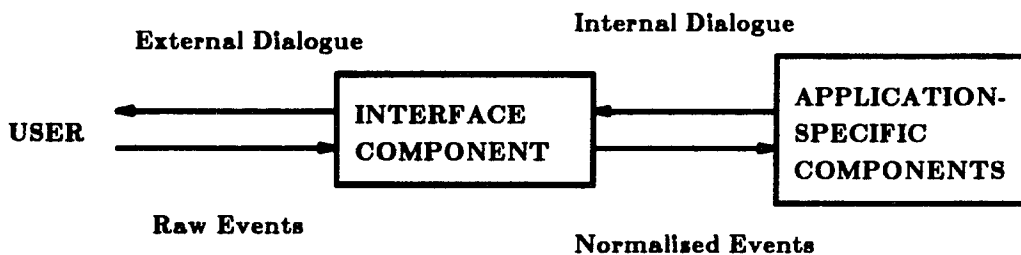


Figure 6.1 A simplified view of an application that has a GUI

The external dialogue occurs between the user and the interface. This dialogue varies among different user-interface systems. This dialogue involves what the user does and perceives in the interaction with the computer. A raw event, such as typing a key or moving the mouse, is the simplest dialogue unit which can have meaning to an application.

The internal dialogue occurs between the interface component and the application-specific components. The raw events generated from the external dialogue are mapped by the interface component to normalized events that the application-specific components will respond to. Normalized events are based on raw events and are sensitive to the interaction context. A raw event may be mapped to different normalized events when placed in different interaction contexts.

Independence of the interface component from the application-specific components is crucial for easy modification and maintenance of a user interface. When this independence is achieved, only the external dialogue and the event mapping in the interface component need be changed to improve or modify the external dialogue style and appearance of an interface. The internal dialogue does need not be changed, and the application-specific components need not be aware of the change.

Depending on the context of the interaction, some raw events may be discarded, and some may be combined to generate normalized events. These operations are performed according to the semantics embedded in the interface component. An application needs to give fast semantic feedbacks to the raw events generated by the end-user's action. Fast feedback is very important for direct manipulation interactions such as dragging an object defined by the application or rubber-banding a line. Both for proper generation of normalized event and for fast semantic feedback, the embedding of high-level semantic capability into the interface component is necessary. However, the amount of embedded semantics capability must be decided carefully. At some level, the developer of the user-interface system must make a clean and logical separation between the application and the user-interface.

#### 6.4.1.1 Low-level Semantics Capability

Lower-level semantics capability in the interface component leads to cleaner separation and stronger interface component independence. As an extreme case, the interface component could pass all the raw events to the application-specific components without performing any interpretation of them. This is called micro-communication (Hartson, 1989). Micro-communication leads to communication overhead in the internal dialogue, and decreases run-time efficiency. Moreover, it jeopardizes application independence. The

application-specific components have to process raw events that should be processed in the interface component. Any change in the external dialogue leads to a change in the application-specific components. This is not consistent with the motivation of separating the interface component from the application-specific components.

#### 6.4.1.2 High-level Semantics Capability

Higher-level semantics capability in the interface component leads to better run-time efficiency. The interface component can consume many raw events and respond quickly to the end-user by its own semantics. Fewer normalized events need be passed to the application-specific components. This is called macro-communication (Hartson, 1989). However, if the level of semantic capability embedded into the interface component is too high, the interface component may become too complex. Moreover, in order to process events and provide proper feedback, this type of interface component has to have extensive knowledge about the application's semantics. Higher semantics capability in the interface component means less independence of the interface component from the application-specific components.

#### 6.4.1.3 The Ideal Level of Semantics Capability

Proper definition of the semantics of the interface component is important to achieve both interface component independence and run-time efficiency. Ideally, the interface component should contain only the semantics that support the interaction functions (i.e., those functions which produce the display and extract valid input). Therefore, it should include the semantics for accepting, parsing, validating, and mapping the raw events to normalized events. It should

also include user prompts, error and confirmation messages, and other displays directly associated with extracting user input. It should only send normalized events to the application-specific components that are meaningful to them.

In some special cases, the interface component may need to share memory and/or data structures with the application-specific components to improve run-time efficiency. However, to achieve interface component independence, coding details associated only with the user-interface should be isolated from those associated with the application's structure and specific components.

#### 6.4.2 Communication and Control

The technique for communication between the interface and application-specific components depends on the run-time control mechanism of the user-interface tools. There are two typical control mechanisms: internal control and external control.

##### 6.4.2.1 Internal Control

If internal control (i.e., application control) is utilized, the application handles the interaction. The application simply calls interface procedures to get events when input is desired. The events are then interpreted by the application according to the interaction context, and dispatched to the appropriate application-specific components to respond to the events. The communications between the interface and the application-specific components are accomplished by the application calling certain procedures of the interface tools. This model is used by many user-interface toolkits such as the Macintosh Toolbox.

A skeleton file is often provided by Toolkits which use internal control. This file contains the basic event polling-loop source code. However, for

applications with a complex user-interface, the polling loop is complicated, and it is difficult to design the application-specific code to fit within it. Even with the availability of the skeleton file, it still takes a considerable amount of time to organize the code of an application to fit within the framework of the skeleton.

#### 6.4.2.2 External Control

Most user-interface development systems use external control (i.e., development system control). The development system handles the interaction. It processes the raw events obtained from either the windowing or the graphics system. When normalized events (generated from the raw events) match with the application's interested in, procedures in the application-specific components will be called to perform operations in response to these events. The most popular communication technique for external control is to use callback functions.

In this approach, the application passes to the development system the pointers to functions that should be called as well as the events that should invoke the functions. The development system then handles all the calls to the callback functions in response to the appropriate events. This is probably the most straightforward and efficient technique (Myers, 1989). The benefit is that the application developer is relieved from handling the interaction process. With external control, the frequency of communication between the interface component and application-specific components is usually less than with internal control. Thus, a better run-time efficiency is achieved.

#### 6.4.2.3 Mixed Control

Finally, there may also be mixed control, where either the development system or the application can be in charge at certain stages of the interaction (Myers, 1989).

#### 6.4.2.4 Use of Shared Memory

In most cases, the memory used by the interface component and the application-specific components is separated. As described above, the communication between the interface component and the application-specific components is accomplished either by the application calling functions in the interface component to get events, or by the interface component calling callback functions in the application to respond to the events. However, the memory used by the interface component and by the application-specific components does not have to be separate. Use of shared memory is another way to accomplish the communication.

In certain cases, such as in a direct-manipulation user-interface, fast semantic feedback requires a frequent communication. The use of shared memory (Myers, 1989), may be better for such cases. In the shared memory approach, both the development system and the application-specific components poll shared data to check for changes, or they are automatically notified of the changes.

### 6.5 Case Study: Macintosh Tools

The Apple Macintosh environment contains several different levels of GUI tools including the Toolbox, the MacApp framework, and the HyperCard environment. Apple has done pioneering work in the development of GUIs.



However, because the Apple Macintosh operating system is a closed system, the general use of these tools is limited. Many developers would like to use the Macintosh Toolbox, MacApp or HyperCard, but they cannot unless they develop their software systems to run only on the Apple Macintosh.

### 6.5.1 Macintosh Toolbox

The Macintosh Toolbox is an example of a user-interface toolkit which uses internal control. The Toolbox offers many handy interface techniques such as menus, scroll bars, and dialogue boxes for building the user-interface. However, to use these techniques, programmers have to handle the user-computer interactions explicitly in the application's code. Figure 6.2 shows a sample procedure for handling an event-loop written in the C language.

As evident from Figure 6.2, the event manager of the Toolbox does not have much semantics capability, and thus programmers are forced to handle details about events. As an example, for the *mouseDown* event, the procedure *do\_mouse\_down* must determine whether this event occurs in a menu bar, in the area for dragging the window, or in some other place. For applications with complex interaction semantics, this type of event processing is too complicated.

An important feature of the Macintosh Toolbox is its resource management utility (Mednieks et al., 1986). Resources are data specifying the static layout of application interfaces and the connections between interaction techniques. A resource utility provides a consistent way for program developers, publishers, and users of the Macintosh computer to specify or modify the interface. It allows the user to change the appearance of the interface of an application to suit his taste without modifying the source code, and it allows the publisher to translate all text in the interface into foreign languages. For the developers, it is even more beneficial. Changes to resources are much easier to make than changes to the code itself. The resource management utility of the

```
event_loop()
{
    EventRecord my_event;
    while (1) {
        GetNextEvent(everyEvent, &my_event);
        switch(my_event.what) {
            case mouseDown:
                do_mouse_down(&my_event);
                break;
            case mouseUp:
            case keyDown:
            case keyUp:
                break;
            case updateEvt:
                do_update(&my_event);
            case activeEvt:
                do_activate(&my_event);
            default: /* some other events */
                break;
        }
    }
}
```

Figure 6.2 Event handling with Macintosh Toolbox

Macintosh Toolbox also allows non-programmers to become involved in the design of the user-interface through the use of a resource editor.

Resources are stored in a standard resource file. Figure 6.3 shows a sample portion of a resource file which specifies a dialogue box (Mednieks et al., 1986). In this figure, the text preceded with double semicolons (;;) is a comment.

As can be seen in Figure 6.3, resources are written in a particular description language, but the level of the language is quite low. The statements are not easy to understand without comments.

#### 6.5.2 MacApp Framework

MacApp is an object-oriented white-box framework that equips programmers with a prefabricated standard Macintosh user-interface for applications. The basic goal of the MacApp is to provide a user-interface framework that: (1) automatically handles the characteristics common to all applications, such as resizing windows, and (2) allows programmers to plug in application-specific details such as the contents of each window (Dodani et al., 1989). There are more than 30 different classes and over 450 methods in the MacApp which handle the default behavior of user-interfaces (Schmucker, 1986). An application programmer can specialize the classes to build a particular application interface by using the inheritance mechanism of the Object Pascal language.

To use the MacApp, an application has to be structured in an object-oriented fashion in terms of the MacApp's object classes. Also, derived classes of certain MacApp classes have to be developed to specialize the application. It is estimated by Schmucker (1986) that the MacApp can reduce application development time by a factor of four or five, and that it can decrease the amount of source code needed for an application also by a factor of four or five.

Type DLOG	::A Dialogue Box
,256	::ID #256
100 100 200 250	::The dialogue's rectangular window
Visible 1 NoGoAway 0	::It is visible, has ProcId, no go away
270	::ID of its item list
Type DITL	::A dialogue's item list
,270	::ID #270
5	::Five items in the list
StatText Disabled	::Uneditable text, not mouse sensitive
20 40 35 180	::The text's rectangular window
A sample dialogue box	::The text
BtnItem Enabled	::A button, mouse sensitive
50 10 70 70	::The button's rectangular window
Resume	::The button's label
ResCItem Enabled	::A control item, defined in a resource
70 10 120 26	::The rectangular window for this control
257	::The resource ID of the control
IconItem Disabled	::An icon
40 150 72 182	::A 32x32 rectangular window
257	::Resource ID of the icon
UserItem Disabled	::An application's own item
80 40 120 230	::The rectangular window it will be displayed in

Figure 6.3 A Macintosh resource file

However, it is clear that MacApp can only be used by experienced programmers. The learning curve for using the MacApp is steep.

### 6.5.3 HyperCard and HyperTalk

Apple introduced HyperCard in the fall of 1987. The HyperCard is a sophisticated environment with elements of database management, object-oriented programming, graphics, and GUIs. It gives non-programmers the ability to manipulate the graphical interface of the Macintosh in a practical way. HyperTalk is an object-oriented language for programming with the HyperCard that is easy to learn and use. The HyperCard environment and the HyperTalk language can be seen as an important software breakthrough. Detailed information about HyperCard and HyperTalk can be found in (Shell, 1989; Weiskamp et al. 1988).

HyperCard provides a set of basic objects for programming. Programs written in the HyperTalk language are called scripts which can be attached to HyperCard objects. Instead of having one large program, an application is divided into many small scripts attached to the objects that compose the application. Objects communicate with each other by receiving and sending messages. Each script is a message sender as well as a message handler. A script is invoked when the object to which the script is attached receives a message.

There are about 47 system messages, and many of these messages are related to the state or position of the mouse. For example, clicking the mouse button at any time causes the HyperCard to send out a sequence of messages starting with the message *mouseDown* and ending with the message *mouseUp* with a number of *mouseStillDown* messages in between. Each command in the HyperTalk language is a message, and applications can also define their own messages.

Figure 6.4 shows a script attached to a Button object which prompts the user to input a text string, and then uses the *find* command to search for the text string (Shell, 1988). Text preceded with double hyphens (--) is a comment. Figure 6.5 shows two dialogue boxes that are invoked by the commands *ask* and *answer* respectively contained in the script. This example illustrates how easy it is to program a user-interface in HyperTalk.

```

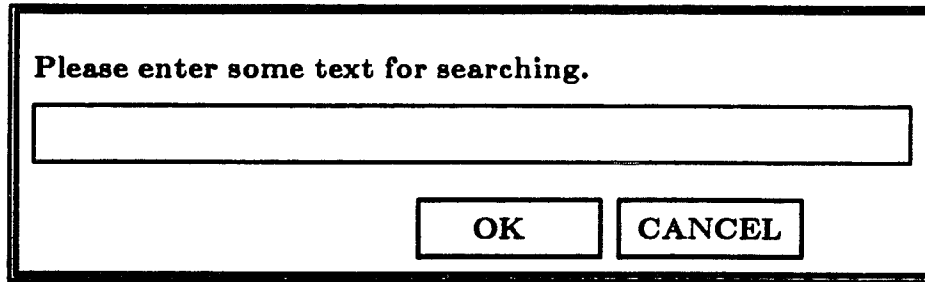
on mouseUp
  ask "Please enter some text for searching."
  if it is empty then exit mouseUp      --Exits if Cancel button pressed
  put it into searchString
  find searchString
  -- If text cannot be found then put up message of explanation
  if the result is not empty then      --That is, if there is an error message
    answer "Can't find the search text in this stack."
    exit mouseUp
  endif
  --To make the Return key do repeated finds
  put "find"&&quote&searchString&quote  --Puts find command in message box
  hide message
end mouseUp

```

Figure 6.4 A HyperTalk script

In this environment, developers of applications can easily represent their ideas and information with words and pictures. The graphical user interface can be implemented without much programming effort. Moreover, because an application is a collection of independent scripts, any script can be enhanced, replaced, or discarded without adversely affecting the operation of the application.

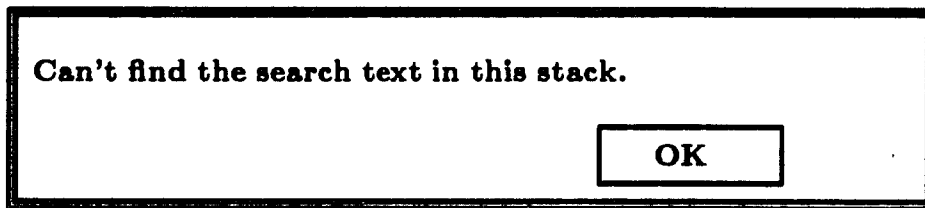
At present the HyperCard is not yet powerful enough. It cannot create applications which can be run outside the HyperCard environment (i.e., the binding of the application and HyperCard is at run-time rather than at link time). It is also a poor environment for team programming, which is essential



Please enter some text for searching.

OK CANCEL

(a)



Can't find the search text in this stack.

OK

(b)

Figure 6.5 Dialogue boxes invoked by the (a) *ask* and  
(b) *answer* commands of HyperTalk

for large projects (Weiskamp, 1988). Moreover, the data structures supported by HyperCard are limited. This is a serious drawback for development of applications involving complicated data structures. However, the HyperCard and the HyperTalk language illustrate the promising features of a graphics-oriented programming environment.

## 6.6 Case Study: The X11 Toolkit

### 6.6.1 Overview

The X11 toolkit (McCormack et al., 1988a, 1988b; Swick et al., 1988) is an object-oriented construction kit built on top of the X Windows System, version 11 (X11). The toolkit is used to write interaction techniques, referred to as Widgets, to organize sets of widget instances into a complete user-interface, and to link a user interface with the functionality provided by an application.

There are three layers in the toolkit: (1) A set of Intrinsic mechanisms to be used by widget programmers to build widgets; (2) An architectural model for constructing and composing widgets, which allows the widget programmer to design new widgets by using the Ininsics and combining other widgets; and (3) A consistent interface for use by application programmers, which is built on top of the toolkit, and includes a set of widgets and composition rules.

A typical X11 Toolkit application consists of three parts: the application, the user-interface, and a link between them. The application is a set of callback functions which the toolkit calls in response to user actions. The user-interface is a tree of widget instances. The link part binds the callback functions and their related data structures to the widgets as the widgets are created. Many applications may use only the existing widgets to build their interfaces. If more specialized user-interface components are needed, widget programmers can create new widgets from existing ones by using the inheritance mechanisms embedded in the X11 Intrinsic.



### 6.6.2 Widgets

The fundamental data type of the X11 toolkit is the widget. Widgets are dynamically created. Every widget belongs to exactly one widget class, which is statically allocated and initialized, and which contains the allowable methods for widgets of that class.

A widget is an object that provides a user-interface abstraction. Physically, it occupies an area on the screen with associated I/O semantics. Some widgets display information, while others are merely containers for other widgets. Some widgets are output only, and do not react to input (i.e., to the user's actions), while others change their display in response to input and can invoke functions that an application has attached to them.

A widget class defines the methods and data that are associated with all widget instances belonging to that class. Methods of a widget class are only callable via widgets of that class. These methods are usually invoked by a set of application callable generic procedures that accomplish operations on widgets by calling appropriate methods. The methods and the data can be inherited by classes derived from that class. A widget class is free to use its base class's methods through the inheritance mechanism of the X11 Toolkit rather than implementing its own code.

Figure 6.6 shows a class hierarchy of the widget set distributed by the *Project Athena* at MIT. The toolkit intrinsic defines four special classes: *Core*, *Composite*, *Constraint*, and *Shell*. All other classes are implemented as direct or indirect derived classes of these classes. The *Core* class contains the definitions of data and methods common to all widgets. It is the root of the class inheritance hierarchy. *Composite* widgets are distinguished by the fact that they can have children. The *Composite* class implements the data and methods for managing children of a *Composite* widget. This includes adding and deleting a child and managing the geometry of children. The *Constraint* class is a

derived class of the *Composite* class. *Constraint* widgets manage the geometry of their children based upon constraints associated with each child. *Shell* widgets hold an application's top-level widgets and allow them to communicate with the window manager.

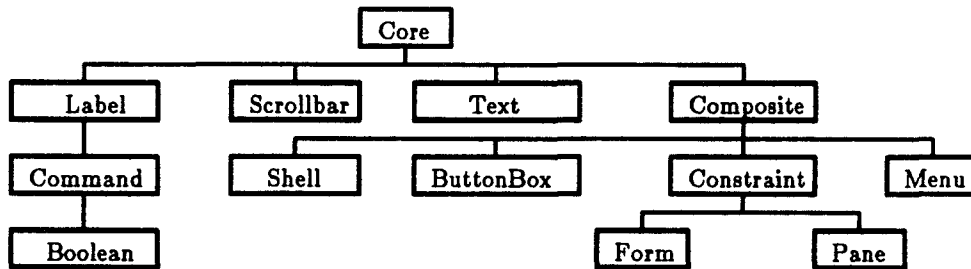


Figure 6.6 Class hierarchy of the widget set distributed by the Project Athena

### 6.6.3 Widget Semantics

The semantics of a widget specifies the mapping (or translation) of event sequences into the behavior of the widget. The simplest example of widget semantics would be to call procedure *Abc* when key *y* is pressed in a particular widget. The semantics of a widget usually are not hard-coded. Instead, the X11 Toolkit provides for a widget default semantics which are overridable by the clients of the widget. Thus, clients (applications or other widgets) may change the semantics of a widget instance when necessary. This mechanism provides a great deal of flexibility in customizing existing widgets.

#### 6.6.4 Event Handling and Callback Mechanism

External interaction control is used in the X11 Toolkit. A typical application consists of startup code followed by an event loop. The event loop is handled in the X11 Event Manager, *XtMainLoop*, which reads events and dispatches them by calling the procedures that have been registered with widgets.

The communication between widgets and their clients, either application functions or other widgets, is established by the clients registering callback procedures to the widgets. A widget can contain one or more callback lists. Each callback list contains at least one callback procedure. Every procedure in a callback list gets called when the condition associated with the list is satisfied.

#### 6.6.5 Critique

The X11 Toolkit provides the basic functionality for building a variety of application user-interfaces. It is extensible and supportive of the independent development of new or extended user-interface tools. Also, it provides an ideal foundation to implement graphical user-interface development systems.

However, as with other user-interface toolkits, the X11 Toolkit is hard to learn and hard to use directly to build an application's user-interface. Figure 6.7 shows the code of an example application, *Goodbye world*, provided by McCormack et al. in their paper (1988).

Shown in Figure 6.8 is the example application. This application first opens and initializes the X server and creates the most-top widget, the "shell" widget of *Shell* class, by calling the function *Xtinitialize*. A *Form* widget, the "box" widget, is then created as a child of the shell. The "box" widget has two children, a "label" widget of class *Label* and a "command" widget of class *Command*. When a mouse button is clicked while the cursor is in the box of the

```

/* include necessary include files */

void Callback(widget, clientData, callData)
Widget widget;
caddr_t clientData, callData;
{
    (void) printf("Goodbye, cruel world");
    exit(0);
}

int main(argc, argv)
unsigned int argc;
char **argv;
{
    Widget toplevel, box, label, command;
    Arg arg[25];
    unsigned int n;

    toplevel = XtInitialize("goodbye", "Goodbye", NULL, 0, &argc, argv);
    box = XtCreateManagedWidget("box", formWidgetClass, toplevel, (Arg *)NULL, 0);
    n = 0;
    XtsetArg(arg[n], XtNx, 10);          n++;
    XtsetArg(arg[n], XtNy, 10);          n++;
    XtsetArg(arg[n], XtNlabel, "Goodbye, world");  n++;
    label = XtCreateManagedWidget("label", labelWidgetClass, box, arg, n);
    n = 0;
    XtsetArg(arg[n], XtNx, 10);          n++;
    XtsetArg(arg[n], XtNy, 10);          n++;
    XtsetArg(arg[n], XtNlabel, "Click and die");  n++;
    command = XtCreateManagedWidget("command", commandWidgetClass, box, arg, n);
    XtAddCallback(command, XtNcallback, Callback, NULL);
    XtRealizeWidget(toplevel);
    XtMainLoop();
}

```

Figure 6.7 An implementation of the *Goodbye world* application with the X11 Toolkit

"command" widget, the callback procedure registered to the "command" widget will be called. This halts the interaction.

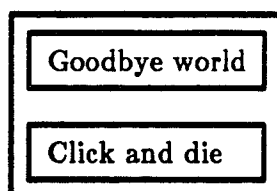


Figure 6.8 The *Goodbye world* application

As shown in this example, the programmer must code the interface explicitly in the application program. Any change to the interface requires recompiling and relinking of the application. Moreover, the user-interface and the application functions are not clearly separated. The statements that link the user-interface to the application-specific functions are intertwined with the construction of the interface. Due to these problems, user-interfaces utilizing the X11 Toolkit directly will be hard to implement and hard to maintain.

#### 6.7 Case Study: GRAFIC/CE88

The graphics package GRAFIC/CE88 (Zhang et al., 1988a) is built on top of the X Window System version 10. It is a modified version of the GRAFIC package developed by Professor D. Anderson at the School of Mechanical Engineering, Purdue University. It was developed as a part of the LineGraph project (Zhang et al. 1988b). Several interaction techniques are provided in GRAFIC/CE88 including *Pull-down Menus*, *Static Menus*, *Dialogue Windows*, and *List Processors*.

Internal interaction control is used in GRAFIC/CE88. In addition to the raw events that the event manager collects from the X Window server, there are several normalized events associated with each interaction technique. For example, the picking of an item in a *Static Menu* generates a `STATICMENU PICK` event.

Applications define their user-interface in resource files written in a simple description language instead of creating the user-interfaces by calling procedures of the GUI tools explicitly. However, the description language in GRAFIC/CE88 is only capable of specifying the static layout of user-interfaces, and thus it can only lead to limited independence of the user-interface code.

Another weakness of the interaction techniques in GRAFIC/CE88 is that their development is based on conventional programming method. Each technique implements a one-of-a-kind user-interface functionality. For example, a *Dialogue Window* can only have three types of items: message, entry, and notifier types. These interaction techniques provide certain fixed styles for user-computer interaction, and are not extendible.

## 6.8 OSF/Motif

### 6.8.1 Overview

The Open Software Foundation has recently released a product OSF/Motif (1990). The development of GUIDES has been performed with only limited knowledge about the details of OSF/Motif. With the current commercial release of Motif (1990), more details about this package are available. OSF/Motif is a graphical user-interface management system consisting of four major components:

1. A graphical user-interface toolkit. This toolkit provides a widget set based on the X Windows (version 11) intrinsics and is similar to the X11 toolkit.

2. A window manager. This manager provides the capability for users to manipulate multiple applications simultaneously on the screen. It is highly customizable so that a user may redefine the window-manager's user-interface and alter other aspects of window-related interactions.
3. A style guide. This guide describes the standard behavior and a set of conventions to ensure a consistent feel on multiple applications.
4. A user-interface language (UIL). The UIL is a specification language for describing the initial state of a user-interface for an application. It enables application developers to describe the presentation characteristics of application interface in a way independent of the actual application code.

In many aspects, the OSF/Motif toolkit is similar to the X11 toolkit except that more widget classes are provided by Motif. Therefore, only the Motif UIL is reviewed here. A comparison of Motif and the GUIDES system developed in this work is made in Chapter 8. A detailed description of OSF/Motif may be found in (Open Software Foundation, 1990).

### 6.8.2 The User-Interface Language

To create the user-interface of an application with the UIL, the interface should be first specified in the user-interface language and stored in one or more UIL specification files. A UIL file can then be compiled by using the UIL compiler to generate a User-Interface Definition (UID) file. UID files are bound with the application code at run time by calling functions of the Motif Resource Manager (MRM).

The Motif user-interface language is a sophisticated specification language. The main purpose of the language is to specify the objects (e.g., menus, buttons, and messages) which compose an interface, and to specify the functions to be called when the objects in the interface change in state due to user actions. It

also has other convenient features such as string concatenation and definition of literal values which may be fetched by the application at run time.

An interface is specified in one or more UIL modules with one module containing the specification of the main window of the interface. A UIL module may be contained in one or more UIL files. A UIL module consists of a series sections denoted by the names of value, identifier, procedure, list, and object, and there can be any number of these sections. It may also contain directives allowing the contents of other UIL files to be included in the module. Figure 6.9 shows a sample UIL module, *hellomotif*. Text that follows an exclamation mark, ("!"), is a comment, and keywords of the UIL are printed in Bold font. This module is from (Open Software Foundation, 1990) and has been modified by the author to show additional features of the UIL in one example. The following sub-sections describe the major features of the UIL used for the definition of a UIL module. These features are illustrated with the *hellomotif* example.

#### 6.8.2.1 Module Header and Header Clauses

The name of a module (*hellomotif* for this example) is declared in the module header following the *module* keyword. The module header may be followed by clauses specifying the version number, the case-sensitivity, etc. of this module. For this sample module, the version number is specified as 'v1.0', and the UIL in this module is specified as case sensitive.

#### 6.8.2.2 The Value Section

A value section consists of the keyword *value* followed by a number of value declarations in the form of *value-name* : *value-expression*. A number of value types are supported in the UIL such as integer, float, string, font, and



```

!+
! A sample Motif module: hellomotif
!-
module hellomotif      ! module header and header clauses
    version = 'v1.0'
    names = case_sensitive
value                  ! value section
    button_x : 15;
    button_y : 60;
identifier             ! identifier section
    call_data;
procedure              ! procedure section
    hellomotif_button_activate(string)
object                 ! object section
    hellomotif_main : bulletin_board {
        controls {
            labels hellomotif_label;
            push_button hellomotif_button;
        };
    };
object                 ! object section
    hellomotif_button : push_button {
        arguments {
            x = button_x;
            y = button_y;
            label_string = compound_string('Hello', separate=true) & 'Motif';
        };
        callbacks {
            activate = procedure hellomotif_button_activate(call_data);
        };
    };
object                 ! object section
    hellomotif_label : label {
        arguments {
            label_string = compound_string('Press button once', separate=true) &
                compound_string('to change label;', separate=true) &
                'twice to exit.';
        };
    };
};

```

Figure 6.9 A sample Motif UIL module

color. After a value name is declared in a value section, it can be referred to in the module in any context where a value can be used. These values may also be fetched in the application code by calling MRM functions. In the sample module shown in Fig. 6.9, two values (i.e., *button\_x* and *button\_y*) are declared.

### 6.8.2.3 The Identifier Section

Identifiers in the Motif UIL provide a mechanism to achieve run-time binding of data items (identifiers) in the application code to names referred to in a UIL module. The name of an identifier in a UIL module is declared in an identifier section which starts with the keyword *identifier*. In the sample module in Fig. 6.9, an identifier *call\_data* is declared. An identifier name may be referred to in the UIL module after it is declared in any appropriate context. An identifier usually represents the value or address of a data item in the application code. This value or address is bound with an identifier name in the UIL module at run time by calling functions of the Motif Resource Manager (MRM). This is discussed in more detail in Section 6.8.3.

### 6.8.2.4 The Procedure Section

The interface defined in a UIL module is bound with the functionality of the application at run time by callback procedures. A typical callback procedure in Motif accepts three parameters: the identifier of the widget to which the callback is registered, the application-specific data (or the client-data), and the widget-specific data. The client-data may be a value of a UIL supported type, or an identifier to a callback procedure, and is specified in the UIL module. A callback procedure is represented in the UIL by a callback name, and the name is bound at run time with the actual address of the callback procedure in

the application code by calling functions of the MRM.

To refer the name of a callback procedure in an object definition, the name has to be declared in a procedure section. A procedure section starts with the keyword *procedure* and contains a number of callback procedure declarations. The type of the client-data to a callback procedure is specified in a procedure declaration by enclosing the type name in parentheses following the procedure name. In the sample UIL module, a procedure *hellomotif\_button\_activate* is declared with the client-data type as *string*.

#### 6.8.2.5 The Object Section

Widget instances (Motif objects) are defined in object sections of a UIL module. Each object section defines one widget instance. It starts with the keyword *object* and contains a sequence of lists that define the attributes, children, and callback procedures for the instance. There are three object sections in Fig. 6.9 defining three widget instances: "hellomotif\_main" of class *bulletin\_board*, "hellomotif\_button" of class *push\_button*, and "hellomotif\_label" of class *label*. The widget instance "hellomotif\_main" is the main window of the *hellomotif* application and the parent widget of the other two widget instances.

The definition of a widget instance may contain an argument list that starts with the keyword *arguments*. The attributes of the instance are specified in the list, and are used when the instance being created at run time. The argument list in the widget "hellomotif\_button" definition shown in Fig. 6.9 contains three attributes: *x*, *y* and *label\_string*.

Children of a widget instance are also widget instances of certain classes and are declared in the control list in the parent widget definition. A control list starts with the keyword *controls* and contains the declaration of each of the child widgets. Two child widgets are declared in the control list of

"hellomotif\_main". The definition of the two child widgets are entered later in the UIL module. A child widget may also be defined in the control list so that it may not be referred to from outside of its parent widget's definition.

In the callback list starting with the keyword *callbacks* and in the definition of a widget, callback procedures are associated with *callback reasons* and specified their client-data. These callbacks should be previously declared in the procedure section of the UIL module. A *callback reason* is related to an event or an event sequence that is meaningful to the widget instance being defined. The definition of the "hellomotif\_button" widget shown in Fig. 6.9 contains a callback list. In the callback list, the callback procedure *hellomotif\_button\_activate* is associated with the callback reason *activate* and is specified the client-data *call\_data*. *call\_data* is an identifier that was declared previously in the identifier section.

### 6.8.3 The Motif Resource Manager

The Motif Resource Manager (MRM) is responsible for creating widgets at run time based on the widget definitions contained in UID files. The UID files are compiled forms of UIL specification files. The role of the MRM in an application is limited primarily to widget creation. The MRM provides interface functions to initialize itself, to register callbacks and identifiers referred to in UIL files, and to create widget objects using the information in the UID files.

After a widget is created by the MRM according to the definition in a UID file, the MRM provides no further services. All the subsequent widget manipulations are done by using functions provided by the Motif Toolkit. Figure 6.10 shows the major steps in the application code to set up an interface with the UIL and the MRM (from Open Software Foundation, 1990).

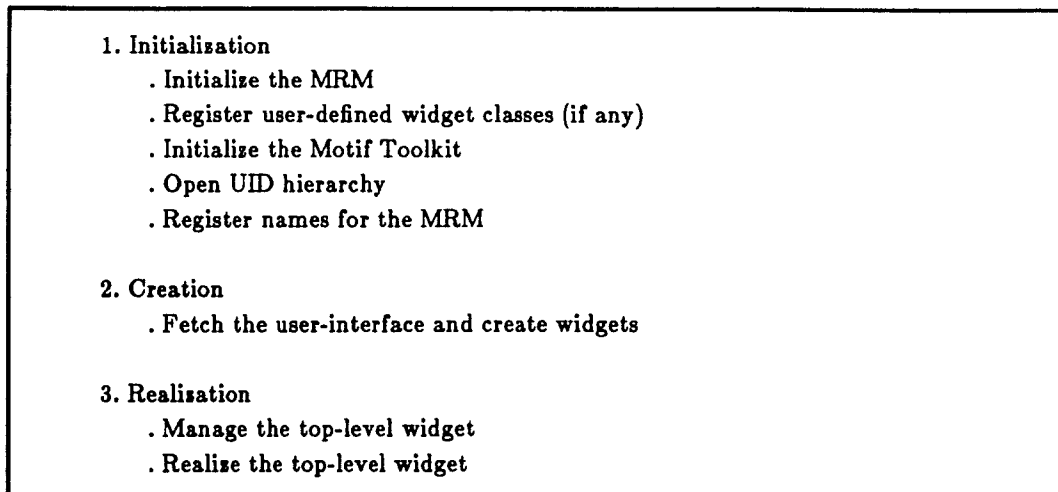


Figure 6.10 Setting up a user-interface specified with UIL

### 6.8.3.1 Initialization

In the initialization step, the application program makes calls to the MRM and Motif Toolkit intrinsic functions to initialize the MRM and the Motif Toolkit, to open the UID hierarchy, and to register names of both callbacks and identifiers. The UID hierarchy is a set of UID files containing the widget definitions for the interface. The addresses of callback procedures and the value of identifiers (value or address of application data) are registered to the MRM with the corresponding names. This is required to resolve name (or symbolic) references in UID files to their run-time values.

### 6.8.3.2 Creation

In the creation step, the application makes calls to the MRM functions to fetch the user-interface. Fetching is a combination of widget creation and child management. By a single call to a MRM function with the top-level widget and

its parent as parameters, all widget instances in the widget tree specified in the UIL files below the top-level widget may be fetched.

#### 6.8.3.3 Realization

The realization of a user-interface defined in UIL files is done in the same way as user-interface created directly using the Motif Toolkit.

#### 6.8.3.4 Other Features

The MRM allows applications to defer fetching certain off-screen widgets until these widgets need to be displayed. This mechanism may be used to reduce the start up time for applications which use many widget instances in their interfaces. The MRM also provides functions for applications to fetch values defined in UIL files. This allows applications to use UIL files as repositories for programming variables to be used to specify the interfaces.

## CHAPTER 7 DESIGN ISSUES IN THE DEVELOPMENT OF GUIDES

In the preceding chapter, the current technologies of graphical user interface tools have been reviewed. In this chapter, an overview of the development of GUIDES, the Graphical User-Interface DEvelopment System for the SESDE, is presented. The usage of GUIDES will be presented in Chapter 8.

Herein, the justifications for the development of GUIDES are discussed first in Section 7.1. The basic requirements and the primary design decisions are presented in Sections 7.2 and 7.3 respectively. Finally, in Section 7.4, the three-dimensional graphics library, HOOPS, is briefly reviewed. GUIDES is built on top of this library.

### 7.1 Justification for the research

Although significant advancements have been made in the technology of graphical user-interface tools in recent years, the technology is still in its youth. During the time period that GUIDES has been developed and implemented (Fall 1988 to Summer 1990), good user-interface development systems are either unavailable (such as the OSF/Motif), or not portable (such as the MacApp and HyperCard, which only operate on Macintosh computers, and the NextStep, which is only available for the NeXT computer).

More importantly, engineering applications often need systems providing both three-dimensional interactive graphics and graphical user interface tools working naturally and compatibly. It has been reported that X Windows will soon gain three-dimensional graphics by merging with PHIGS - the

Programmer's Hierarchical Interactive Graphics System (Langa, 1989). PHIGS implements an internationally accepted standard for three-dimensional graphics. It is not clear, however, when this package will be available. Furthermore, the PHIGS standard, like many standards, is evolving over time as it is enhanced with new features. It is not feasible to wait for systems to evolve that will facilitate optimally the development of interactive graphics applications.

Graphical user-interfaces are one of the most important features of Computer-Aided Engineering software. Research is needed to investigate and to develop improved methods for using three-dimensional graphics and user-interaction techniques for the next generation of engineering software. The SOCRATES project (Ingraffea et al., 1988) has performed pioneering work in this area. Several general interaction tools have been developed and utilized in the SOCRATES software, such as lists and numerical keypads. There are also many well-known interaction techniques (such as popup menus) that are provided by almost every windowing system. It is desirable to reproduce these tools in a manner such that they can be: (1) implemented separately from any application-specific software; (2) utilized and handled consistently and easily in any engineering software; and (3) ported easily without any major dependence on the specific hardware or operating system.

The main goal of GUIDES is to facilitate the development of engineering applications through rapid prototyping and testing of user-interfaces. Another goal is to provide a crucial layer between engineering applications software and evolving software technology. Since many adjustments to utilize new computer graphics technology can be made within GUIDES, applications utilizing this package will be more adaptable to new computer technology. Changes in the graphical user-interface can be made with minimal effects on the application-specific code. Applications will be able to take advantage of advances in hardware, operating systems, and graphics libraries in a more effective way.



To this end, existing GUI systems such as the Macintosh GUIs and the X11 toolkits have been carefully studied. Attempts have been made to integrate their best features and to avoid their weaknesses in the design of GUIDES. These aspects are discussed in the following sections.

## 7.2 Basic Requirements

The following basic requirements have been established for GUIDES:

1. GUIDES should lead to a better separation between the user interface and application-specific components.
2. GUIDES should not force applications to follow any specific programming style or programming language.
3. GUIDES should provide a reasonable set of interaction techniques. Also, it should be extensible such that it supports the creation of new interaction techniques.
4. GUIDES should work with a three-dimensional graphics package naturally and compatibly.
5. GUIDES should be easy to use by both expert and non-expert programmers.
6. GUIDES should be portable. It should not be restricted to certain hardware or operating systems.

According to these requirements, several design decisions have been made as described in the next section.

## 7.3 Design Decisions

### 7.3.1 Black-Box versus White-Box Framework

As discussed in Section 6.3.2, the strength of a white-box framework is its flexibility. The interaction techniques defined in a white-box framework can be tailored to suit the particular interface styles of different applications. The weakness is that intimate knowledge of the underlying implementation is required to use a white-box framework. Primarily the white-box framework is useful only for expert programmers.

In contrast, the strength of a black-box framework is ease of use. No knowledge of the underlying implementation is required. Also, the use of a description language to specify the user-interface of an application leads to a better separation of the interface and application-specific components. The weakness is that applications can only be provided a limited number of types of interaction techniques to build their own interface. Application programmers also need to learn the description language in order to specify the user-interface.

In a university environment, the potential users of GUIDES are students and faculty in engineering areas. These users are experts in certain application areas, but typically they are not experts in programming. In fact, they need not be. However, computation related research and instruction are gradually becoming more and more software intensive. Therefore, software such as GUIDES is essential for the development of graphical user-interfaces in a university setting.

Ease-of-use is considered to be more important than flexibility in the design of GUIDES. Therefore, a black-box framework has been chosen for the design. To avoid the weaknesses associated with the black-box approach, a reasonable set of interaction techniques is provided by the framework of GUIDES. These techniques satisfy the interaction requirements for most applications. Whenever a particular new interaction technique is needed, it can

be developed by GUIDES programmers based on the existing techniques of GUIDES. The syntax of the GUIDES description language is simple and clearly defined. However, a construction set should be provided to help application programmers construct the user-interface interactively. Such a construction set will be developed in the future work on GUIDES and is not included in the present work.

### 7.3.2 Windowing-System versus Graphics-System Basis

To obtain a system which can handle both three-dimensional interactive graphics and interactive user-interface tools naturally and compatibly, there are three choices: (1) developing a user-interface development system based on an existing three-dimensional graphics package, (2) developing a three-dimensional graphics package for an existing user-interface development system, and (3) utilizing both an existing three-dimensional graphics package as well as an existing user-interface system. The third choice is not taken because it is difficult, if not impossible, to make an existing graphics package and an existing user-interface system work together well. The decision is therefore between the first two choices. The technology for three-dimensional graphics systems is more mature than that for user-interface development systems. In fact, several sophisticated three-dimensional graphics packages are available commercially: HOOPS (Wiegand, 1988) by Ithaca Software, several implementations of the PHIGS (Brown, 1985), and others. GUIDES is built on top of HOOPS such that applications utilizing GUIDES can take advantage of both three-dimensional interactive graphics and interactive user-interface tools naturally and compatibly.

Moreover, building GUIDES on the HOOPS commercial graphics package has another important benefit. HOOPS is supported on equipments ranging from the IBM/PC/AT and Apple Macintosh II to high-end workstations such as

the Silicon Graphics IRIS. Therefore, GUIDES does not have to interface to any specific windowing system by itself. Instead, its interface with different windowing systems is via HOOPS. GUIDES can be ported to any specific hardware and run on any operating system where HOOPS is available. However, GUIDES is not heavily dependent on HOOPS. Carefully defined policies are enforced in the implementation of GUIDES to minimize the dependency of the system on HOOPS. GUIDES could be ported with relative ease to another graphics package which has similar functionality to HOOPS (e.g., a PHIGS package).

### 7.3.3 Design Methodology

To achieve the goals of reusability and extendibility, an object-oriented approach is used to design the interaction techniques of GUIDES. The interaction techniques of GUIDES are referred to either as agents or graphical utilities. Agents provide generic user-interface tool abstractions, such as *Buttons*, *Menus*, and *Dialogue Windows*. Graphical utilities are tools designed to accomplish specific tasks such as drawing and managing X-Y plots. Graphical utilities utilize agents to accomplish their functionality.

Agents are identified as the objects in the design of GUIDES. An agent is a software object providing an interaction technique abstraction. The GUIDES agents will be discussed in more detail in Chapter 8. Design and implementation details of GUIDES agents have been reported in (Zhang et al, 1990).

It should be mentioned that although GUIDES is an object-oriented design, applications are not forced to use the same design methodology. Agents and their methods are enveloped by groups of application-interface functions which are conventional functions. Because the agent instances constituting a user-interface are defined by using the description language of GUIDES, no

application interface function is needed to create agent instances "on the fly." This greatly reduces the number of application interface functions. The application interface functions of the agents are used for editing agent instances, for configuring agent instances, and for retrieving event information from agent instances.

#### 7.3.4 Internal versus External Control

External control (i.e., control by GUIDES) has obvious advantages over internal control because application programmers are relieved from handling the interaction process. Therefore, the complexity of application code is reduced. However, in certain cases, internal control and mixed-control may lead to a higher run-time efficiency. Moreover, the internal control approach shows the event handling process explicitly in the application code. This may be important to accomplish certain interactive techniques such as rubber-banding a line. Therefore, even though external control is often the best approach, both external control and internal control are supported.

For external control, the communication between GUIDES and applications is via callbacks. When an application does not register callbacks to an agent instance, control is returned to the application when that instance is invoked. This results in internal control. When the application does not register callbacks to GUIDES at all, the application would use internal control completely. If callbacks are registered for some instances and are not registered for others, this results in mixed control. Another form of mixed control is that callback functions can call an interface function to obtain the next event from the event queue. This type of mixed control is also available in GUIDES.

### 7.3.5 Agent Semantics

As discussed in Section 6.6.3, the X11 Toolkit provides the capability for its applications to override the default semantics of X11 Widgets. This leads to flexibility in customizing a X11 Widget for different interaction techniques. However, it is not a trivial task to take advantage of this flexibility, and the price paid for this flexibility is complexity. Moreover, it is not necessary to override the semantics of interaction techniques if these semantics are flexible enough, and if there exists a reasonable set of interaction techniques to fit the requirements of most applications.

The semantics of GUIDES agents are fixed. An agent behaves as specified by its class protocol. This is true whether it is used as a standalone interaction technique or as a component of an interaction technique. However, to provide flexibility in customizing agents to build complex interaction techniques, slots are provided that clients of agents can fill in. This aspect is described bellow.

Agents that accept user input can be displayed in more than one mode. They can change their different mode in response to associated user actions. Callback lists are maintained by the agent instances for each mode change. Each agent instance maintains one or more callback lists. These lists are identified by a specific name and are defined in the class protocol of that agent. The callback lists of an agent instance, in fact, are pre-defined slots by which the semantics of the agent are connected to the application functionality or the semantics of other agents.

A callback list of an agent instance is associated with a certain event or a set of events which cause a change in the display mode of the instance. A list may contain one or more callbacks or it may be empty. Once an appropriate event related to the agent instance occurs, the instance changes its mode, and then calls the callbacks in the list associated with that event, if the list is not empty.

For example, depressing a mouse button while the mouse pointer is in a *Button* agent instance will cause the *Button* to change to the *selecting* mode. The callbacks in the *selecting* callback list will be called if there are any. The *Button* does not care who registers the *selecting* callbacks to it. If an application registers the callback, the callback will perform the application specific functionality. If another agent instance, which the *Button* is a component of, registers the callback, the callback will accomplish the semantics of that instance.

Therefore, although GUIDES agents have fixed semantics, they are still flexible enough to be either used as standalone interaction techniques or connected by the callback mechanism to form more complicated interaction techniques.

### 7.3.6 Language Binding

The C language is clearly better than FORTRAN for interactive graphics applications. However, since FORTRAN is still the major language for engineering applications, the application interface functions of GUIDES should be made callable from both C and FORTRAN. In order to achieve this, data structures or pointers to data structures are avoided as arguments to application interface routines unless other functions which accomplish the same functionality without pointers or data structures are provided for FORTRAN programs. Returned values from all interface functions are integers. However, a complete FORTRAN interface has not been made available in the present work on GUIDES.

## 7.4 The HOOPS Graphics Library

HOOPS (the Hierarchical Object-Oriented Picture System), which is marketed by Ithaca Software (Wiegand, 1988), is a powerful and portable three-dimensional interactive graphics package. Since versions of HOOPS are supported by Ithaca Software under many operating systems such as DOS, the Macintosh operating system, UNIX, and VMS, an application utilizing HOOPS should port readily to any of these systems. At present, several HOOPS-based three-dimensional CAD systems are under development (Kliwer, 1989).

Because graphics applications must maintain a picture database storing the drawing primitives and their drawing attributes, HOOPS is first a database system. It stores information about which objects to draw, where they should be displayed, and how they should be rendered. The basic unit in the database is called a segment. A segment is a collection of attributes, geometry, and other segments, grouped together as an object. Geometry refers to drawing primitives including lines, polygons, text, and markers, which are the basic building blocks for a picture. Attributes (e.g., colors, patterns, and projection methods, etc.) describe how to display the drawing primitives.

Each segment may also contain other segments, called subsegments of that segment. This results in a hierarchical structure. At the top of the hierarchy, there is a special segment known as *?Picture* which refers to the current graphics device. Any segments under *?Picture* will be automatically displayed or redisplayed unless their visibility is turned off. Each segment can be associated with a window where the drawing primitives in that segment are displayed (however, a segment is not required to contain a window). The window of the *?Picture* segment corresponds to the full screen by default. The window of a subsegment is described as a portion of the window of its parent. A segment along with its primitives and subsegments can be manipulated as a whole (such as in scaling and resizing). The attributes of a segment can be inherited by its subsegments such that any change in an attribute of a segment



will affect all its subsegments unless the attributes are also specified for the subsegment.

Another important feature of HOOPS is the *?Include Library*. This library is a separate branch of the HOOPS segment tree. It is not visible, and segments defined under it will not be displayed. However, these segments can be included one or more times by segments under *?Picture*. Therefore, the objects defined in segments under the *?Include Library* can be displayed in different views simultaneously without duplicating the definition of these objects. Figure 7.1 shows the hierarchical structure for an application which displays a car in two views.

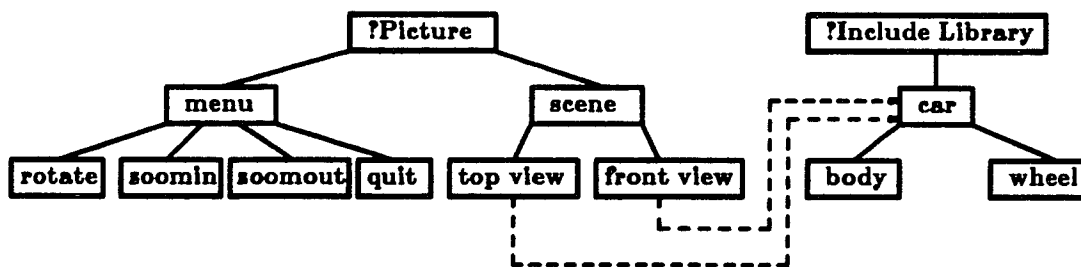


Figure 7.1 Hierarchical structure of HOOPS segments for an application which displays a car in two views

HOOPS provides only limited support for building graphical user interfaces. It identifies five basic types of raw events as listed in Figure 7.2. Also, it maintains an event queue. It is worthy to note here that a long integer, called the user value, can be stored within a segment. This value can be retrieved from HOOPS when an event occurs in that segment. This allows HOOPS applications to relate a specific piece of application data or a specific application procedure to a segment.

Event Type	Circumstance
Wakeup Event	The process has slept for a specific time or until a certain event occurs.
Button Event	A key on the keyboard is typed.
Location Events	Location events are detected in raw device coordinates.
"v"	Mouse just went from "no button down" to "at least one button down"
"^"	Mouse just went from "one or more button down" to "no button down".
"*"	Location of mouse pointer has changed with one or more button down.
"O"	Location of mouse pointer has changed with all buttons down.
Selection Events	Selection events are detected in window coordinates of segments.
"v"	Mouse just went from "no button down" to "at least one button down"
"^"	Mouse just went from "one or more button down" to "no button down".
"*"	Location of mouse pointer has changed with one or more button down.
"O"	Location of mouse pointer has changed with all buttons down.
String Event	A text string ended with a carriage return is typed.

Figure 7.2 Raw events identified by HOOPS

High-level interaction techniques are not provided by HOOPS. Applications not using GUIDES have to build interaction techniques and handle user-application interactions explicitly in their code. For example, the segment "menu" in the application shown in Figure 7.1 would contain the user-interface techniques. It would represent a static menu which is hard-coded in the application. In many existing HOOPS applications, a large percentage of code has to be devoted for building interaction techniques (such as static menus) and handling interactions explicitly. As a result, duplication of code within an individual program, and among programs associated with different projects cannot be avoided. The user-interfaces code implemented directly with HOOPS is hard to design, modify, and maintain. GUIDES solves these problems.

## CHAPTER 8 DESCRIPTION OF GUIDES

An overview of the development of GUIDES is presented in the preceding chapter. In this chapter, major features and use of GUIDES are presented. The architecture of GUIDES is described in Section 8.1. The GUIDES Event Manager and Callback Manager are discussed in Sections 8.2 and 8.3. GUIDES Agents and Graphical Utilities are described in Sections 8.4 and 8.5. The GUIDES Description Language is outlined in Section 8.6, and a complete example application is illustrated. Finally, a comparison of GUIDES and OSF/Motif is made in Section 8.7. More detailed documentation on the use of GUIDES is given in the GUIDES Reference Manual (White et al., 1990), and design and implementation details are reported in (Zhang et al., 1990).

### 8.1 Architecture of the System

GUIDES is composed of four major components: the Description File Parser, the Callback Manager, Interaction Techniques (agents and graphical utilities), and the Event Manager. When external control is used, an application utilizing GUIDES will consist of three parts: (1) Callback Procedures which implement the functionality of the application, (2) Description Files which present the specification of the user-interface, and (3) an Application Initialization component which initializes GUIDES and the application and starts the interaction process. Figure 8.1 shows this conceptual model of GUIDES and the application, where solid lines with arrows pointing from one component to another indicate the functions in that component calling functions in the other component.

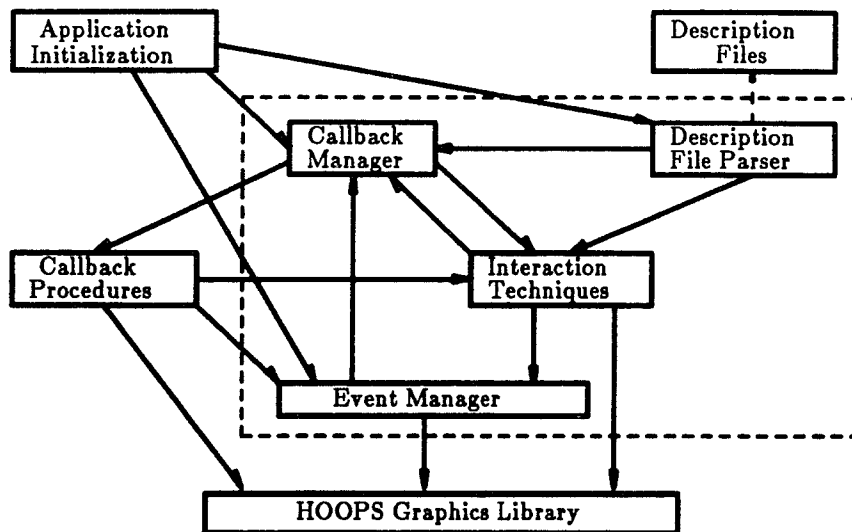


Figure 8.1 The conceptual model of GUIDES

The Callback Manager maintains a list of callback procedures registered by both applications and interaction techniques. A callback procedure may be represented by a callback name in the description file. The Description File Parser parses description files passed to it by the Application Initialization component. According to the specification in the description files, the Parser creates the interaction techniques which the user interface is composed of, and establishes the connections between interaction techniques and between the user-interface and the application. The Event Manager reads raw events from HOOPS, interprets the raw events, and then invokes the appropriate callback procedures in either the interaction techniques or the application program. These components are discussed in more detail in the following sections.

## 8.2 Callback Manager

Callback functions bind the Event Manager, the agent instances, and the application code together forming a complete mechanism for handling interactions between the user and the application program. Callback functions used by GUIDES must be defined as functions returning an integer. The integer value returned must be either one of the two pre-defined GUIDES constants *GS\_CONTINUE* or *GS\_EXIT*. *GS\_CONTINUE* should be returned if the application wishes for GUIDES to continue the interaction control, and *GS\_EXIT* should be returned if the interaction control should be returned to the application. The general form of a callback function is shown in Figure 8.2.

```
int
callback(client_data, p_event_record)
Gs_ClientData client_data;
Gs_Event      p_event_record;
{
    (implementation details)
}
```

Figure 8.2 The form of GUIDES callback functions

In Fig. 8.2, *callback* is the name of the callback function. *client\_data* is the client data which represents or may be used to access the data in the application code required by the callback function. The client data is registered to GUIDES together with the associated callback function and is passed to the callback function when the function is invoked. *p\_event\_record* is a pointer to the record of the event by which the callback function is being invoked. The callback function may obtain detailed information about the event by calling GUIDES interface functions with this pointer as an argument.

In GUIDES, a callback function pointer and its associated client data are identified by a callback name. The callback name is a string which may or may not be the same as the name of the callback function. As a result, a particular callback function in the application-specific code may correspond to one or more callback names (and thus one or more different callbacks), and each of these different callbacks may be associated with different client data.

An application must register its callbacks to GUIDES at run time. A record is maintained by GUIDES for each callback registered to it. This record consists of three pieces of information: the callback name, the pointer to the callback function, and the client data. This record is a data structure which is used by the application when registering callback records to GUIDES. The callback records are maintained in a callback table. The Callback Manager provides interface functions for installing callback records to the table, for redefining a callback, and for looking up a callback record by its name from the table.

A callback is associated with a certain event in an *Application Window* agent (see Section 8.4.6.1 for a description of the *Application Window* Agent) by calling interface functions of the Callback Manager. A callback is associated with a certain event in an agent other than *Application Window* by referring the callback name in the definition of the agent in a description file.

### 8.3 Event Manager

The Event Manager is an essential component of GUIDES. It manages the communications between GUIDES agents and HOOPS as well as communications between the application and HOOPS. It does not distinguish between the type of its clients. The clients (i.e. the code units which use the Event Manager) include both interaction techniques and application-specific components. Both clients are treated in a same manner.

The major functions of the Event Manager are: (1) to expand the number of event types provided by HOOPS to include a number of normalized event types, (2) to manage the asynchronous queuing of events, and (3) to process the callbacks associated with any particular event. Event manager interface functions are provided for defining the types of events that the application is interested in for any segment of the HOOPS database, for extracting information about a current event, and for saving an event sequence to a file or reading and responding to a series of events from a file.

### 8.3.1 Processing of Raw Events

The Event Manager is invoked in an application code by calling the function *GS\_EventManager*. The first task of the Event Manager is to dequeue and interpret raw events generated by user actions known to HOOPS. The Event Manager interprets the raw events and generates normalized events which are higher level than the raw events. These normalized events are called basic GUIDES events. Figure 8.3 lists the 13 basic GUIDES events identified by the Event Manager. Several of the basic GUIDES events are actually identical to particular HOOPS events. The Event Manager also queries and saves into an event record whatever information is available from HOOPS about the current basic event (such as the segment in which the event has occurred).

To be notified of the occurrence of certain basic events within their segments, clients must inform the Event Manager of the basic events they are interested in. They can also register callback functions associated with these events. The Event Manager maintains a segment-events list and stores the events each HOOPS segment is interested in and the callbacks associated with the segments. Segment user values are employed as indices of corresponding entries in the list for fast processing. Any event the clients are not interested in will be discarded by the Event Manager without notifying the clients.



Event Type	Circumstance
PAUSE	The process has slept for a specific time or until a certain event occurs.
KEY_STRIKE	A key on the keyboard is typed.
BUTTON_DOWN	A mouse button is pressed.
BUTTON_UP	A mouse button is released.
BUTTON_UP_MOTION	Mouse pointer is moved with all buttons released.
BUTTON_DOWN_MOTION	Mouse pointer is moved with at least one button pressed.
BUTTON_DOWN_STILL	No further event is detected after a BUTTON_DOWN.
BUTTON_CLICK	A BUTTON_DOWN followed by a BUTTON_UP event.
BUTTON_DOWN_EXIT	Mouse pointer is moved out the current segment/window with a button pressed.
BUTTON_DOWN_ENTER	Mouse pointer is moved into a segment/window with a button pressed.
BUTTON_UP_EXIT	Mouse pointer is moved out the current segment/window with all buttons released.
BUTTON_UP_ENTER	Mouse pointer is moved into a segment/window with all buttons released.
STRING_INPUT	A text string ended with a carriage return is typed.

Figure 8.3 Basic events of GUIDES

### 8.3.2 Handling of Basic GUIDES Events

The second task of the Event Manager is to handle the processing associated with the basic GUIDES events that clients are interested in. If a client is interested in the current event, the Event Manager must first check if a callback function has been registered with the event.

If no callback is registered, the Event Manager returns from the *GS\_EventManager* function to the application. A pointer to the current event record, generated by the Event Manager in its first task, is returned through the parameter list of *GS\_EventManager*. The event information stored in the current event record can be retrieved by clients through interface functions of the Event Manager. The application, of course, can call *GS\_EventManager* again to pass the interaction control back to the Event Manager.

If a callback function has been registered with the current event, the Event Manager calls the function and passes a pointer to current event record and the client data as arguments. A callback function is expected to return either of the flags *GS\_CONTINUE* or *GS\_EXIT* as discussed in Section 8.2. If *GS\_CONTINUE* is returned, the Event Manager will continue its task to dequeue and process the next event from HOOPS. If *GS\_EXIT* is returned, the Event Manager will return the interaction control to the caller (i.e., the application). In this case the Event Manager expects the callback function to store its associated event record in an event register, a block of memory accessible by both the Event Manager and its clients. Although the Event Manager does not distinguish its clients, GUIDES does not allow application callback functions to access the event register. Only the agents know and have the access to the event register.

### 8.3.3 The Event Register

If the event register is not empty when the flag *GS\_EXIT* is returned from a callback function, the Event Manager will pass the event information in the register to the application as the current event record. If the event register is empty, the event record generated by the Event Manager (which contains the information about the current event viewed by the Event Manager) will be stored. A pointer to the current event record is passed back to the application via a parameter of *GS\_EventManager*.

Only the clients who are GUIDES agents have any knowledge of the event register. The event register is hidden from actual applications. Each type of agent may generate a particular type of Agent Event and a particular type of event record. A typical example is that a *Static Menu* may generate a *STATICMENU\_EVENT* when a mouse button is clicked while the mouse pointer is inside an item of the menu. The event record generated by the *Static Menu* stores the event identifier, *STATICMENU\_EVENT*, together with the name of the *Static Menu* instance and the identifier of the item in the menu just picked. The *Static Menu* agent puts its own event record in the event register and returns *GS\_EXIT* to the Event Manager upon certain conditions. These conditions depend on the particular semantics of the agent. The agent semantics are specified in detail in the GUIDES Reference Manual (White et al., 1990). The application can retrieve event information generated by an agent only through application interface functions provided by the class of that agent.

### 8.3.4 Grabbing of Events

In some situations, events need to be grabbed. In other words, incoming events should be sent only to a particular set of segments. A typical example is that, when a *Dialogue Window* becomes active, events need to be grabbed to the segments of the agent instances in the *Dialogue Window*. The dialogue session

must be completed before any other interaction can proceed. The Event Manager maintains a stack containing the set of segments that can accept events at a certain time. Each time an event grabbing process is invoked, an entry is pushed on top of the stack. This entry contains a new set of segments that events should be grabbed to. When an event grabbing process is finished, the top entry in the stack is popped (removed) from the stack. The Event Manager sends events only to the segments listed in the top entry of the stack.

### 8.3.5 Queueing of Events from a File

Another feature of the Event Manager is that it can store a sequence of events to a file. Also it can process a sequence of events stored in a file, just as if those events were generated by the end-user's action and dequeued from HOOPS. This capability enables: (1) automatic demonstration of an application program, i.e., an application program can step through a certain sequence previously performed by a user; and (2) an automated type of command driven interface, i.e., once a user has issued a certain command sequence once, he can save the corresponding sequence of events to a file, and whenever it is desired to execute that sequence again, the file can be utilized to automatically perform the sequence.

## 8.4 GUIDES Agents

The basic elements of which GUIDES is composed are referred to as agents. An agent is a combination of a window displayed on the screen at a certain time, and its associated input and output semantics. Some agents do not have their own I/O semantics and are just used as containers for other agent instances. An individual interaction technique can be constructed by an individual agent instance or a combination of several instances.

### 8.4.1 Agent Classes and Agent Groups

Agents are identified as objects providing user-interface abstractions. An agent is composed of its private data and its private and public functions which manipulate the data. Each type of agent is identified as an agent class. An agent class serves as a "template" by which the agent instances (i.e., the objects) corresponding to that class can be created. Agent instances of the same class have a common set of properties and possess the same set of methods.

There are also abstract classes in the design of GUIDES. Abstract classes can never have any instances. They are used only for building the inheritance hierarchy. There are three abstract classes in GUIDES: the *Basic* class, the *Composite* class, and the *Restricted Composite* class. All agent classes are derived from a particular abstract class.

The hierarchical inheritance structure of GUIDES is shown in Figure 8.4. The agent classes shown in this figure are those currently developed in the present work. Additional agent classes may be developed in future work.

Agent classes derived directly from the same abstract class are said to belong to the same agent group. Therefore, there are also three agent groups in the design of GUIDES. These are the simple agent group, the composite group, and the restricted composite group.

#### 8.4.1.1 Simple Agent Group

Agents in the simple agent group are derived directly from the *Basic* abstract class. These agents are used to accomplish the most basic functions for building user-interfaces. This group contains the following classes: *Message*, *Button*, *Entry*, *Toggle*, and *Application Window*. These agent classes are described further in Section 8.4.6.1.

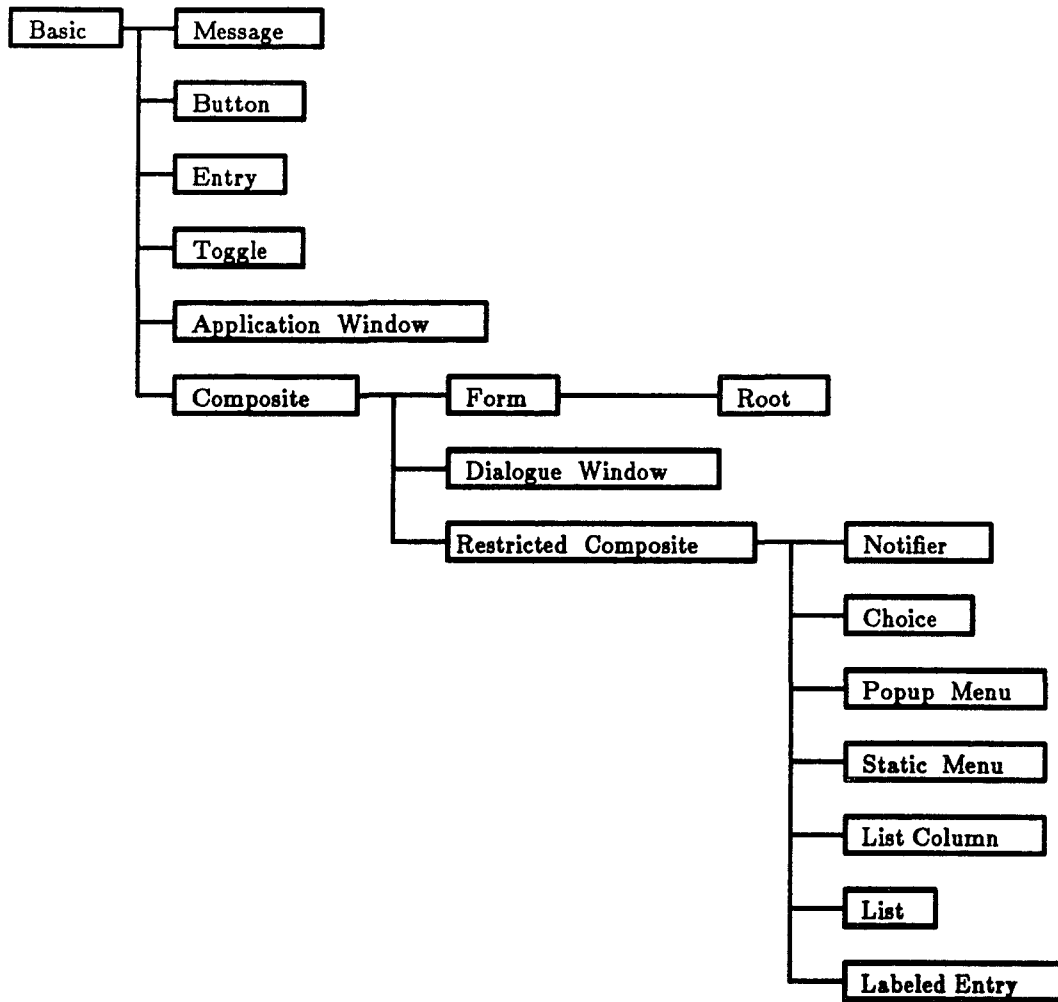


Figure 8.4 Agent class inheritance hierarchy of GUIDES

#### 8.4.1.2 Composite Agent Group

Composite agent classes are derived from the *Composite* abstract class, which is derived in turn from the *Basic* abstract class. Composite agents are simply containers to compose one or more simple, composite, or restricted composite agent instances as a unit which can be defined and manipulated as a whole. A composite agent instance is called the parent of the instances of which it is composed. The agent instances combined to make a composite instance are referred to as the children of the composite instance, and are said to be instanced by the composite instance. Three agent classes belong to this group: *Form*, *Root* and *Dialogue Window*. These agent classes are described further in Section 8.4.6.2.

#### 8.4.1.3 Restricted Composite Agent Group

Agents of the restricted composite group are derived from the *Restricted Composite* abstract class, which is derived in turn from the *Composite* abstract class. The following agent classes belong to the restricted composite group: *Labeled Entry*, *Notifier*, *Choice*, *Static Menu*, *Popup Menu*, *List Column*, and *List*. These agent classes are described further in Section 8.4.6.3

Restricted composite agents are special types of composite agents in that:

1. Component instances of a restricted composite instance are restricted to certain agent classes and are created by the restricted composite instance. Conversely, the children of a composite instance can be of any agent class, and the composite instance does not create its children. For example, a *Static Menu* is a restricted composite agent built from one or more *Button* instances referred to as the items of the *Static Menu*, and the *Static Menu* creates these *Button* instances.

2. The geometry of the component agent instances (location and size) within a restricted composite instance are managed by the restricted composite agent, whereas the geometry of the children of a composite agent instance must be explicitly specified when they are instanced. For example, the *Buttons* in a *Static Menu* instance will be packed in the window of the *Static Menu* with the same size, and in the form of a column, row, or two-dimensional array.

#### 8.4.2 Agent Composition

Actual agent instances created in a user-interface are linked together to form a hierarchical tree structure of agent composition relationships. At the top of the tree is an object of the *Root* agent class derived from the *Form* agent class. The *Root* object is the parent of all the agent instances defined in the application interface.

There are two general rules in the design of GUIDES for building composite agent instances. The first rule is that a composite agent instance must hold a list of pointers to its children so that it can send messages to them. However, the children generally should know nothing about their parent. The second rule is that a composite agent instance does not have any control over the appearance and I/O semantics of its children. The above two rules allow standardization of the connections between parents and children. The communication between a parent instance and its children is by sending messages. This communication is flexible such that: (1) it should be easy to derive other composite agent instances; and (2) the children of composite agent instances can be children of other composite instances, or an agent instance may have multiple parents.

The rules for building composite agent instances are also valid for building restricted composite instances. However, since a restricted composite instance



creates and manages the geometry of its own children, it may register its own functions as callbacks to its children to accomplish its own I/O semantics. These callback functions would be called during interaction to perform the functionality of the composite instance. Since a restricted composite instance can install its own callbacks to its children, the children of restricted composite instances cannot be a child of any other agent instance.

The compositing hierarchy for the agent instances should not be confused with the inheritance hierarchy for the abstract and agent classes as shown in Figure 8.4. A composite agent instance is composed of a group of agent instances, each of which constitutes a part of the composite instance. The composite instance is the parent of the agent instances (objects) that are its children. It represents the children as a group such that the group can be manipulated as a whole. The parent object and each of the children objects possess the properties and methods of their own respective classes. All of the objects inherit their properties and methods from the base classes of their class.

### 8.4.3 States of Agent Instances

An agent instance may exist in any one of the following four states:

1. **Created state:** The agent instance has been defined.
2. **Realized state:** The agent instance has been instanced by at least one other agent instance.
3. **Mapped state:** The window of the agent instance is being displayed on the screen, and the instance is ready to accept events.
4. **Active state:** The agent instance has received an event and the interaction between the user and the instance is in progress.

Generally, an agent instance must be created before it can be realized, it must be realized before it can be mapped, and it must be mapped before it can be active.

With the exception of the *Root* agent instance, there is no limit to the number of agent instances that can be utilized by an application to build its user-interface. However, an application has only one *Root* agent instance. This instance is automatically created and mapped by GUIDES during the initialization stage.

#### 8.4.3.1 Creation of an Agent Instance

An agent instance can be created in one of two ways: (1) it may be defined explicitly in a description file; or (2) it may be created as a child of any agent instance belonging to the restricted composite group.

#### 8.4.3.2 Realization of an Agent Instance

The location and size of an agent instance's window are specified in terms of the parent's window coordinates at the realization stage. If an agent instance is created by its parent (or the parent is of a restricted composite agent class), it can have only one parent and can be realized only by its parent. If an agent instance is defined explicitly in a description file, it may be realized (or instanced) by one or more composite agent instances. Thus, such an agent instance may become a child of any composite agent instance which realizes it, and it may have multiple parent. However, it can be displayed on the screen in only one of its parent's windows at any one time. Each of its parents holds the location and size of the instance in the parent's window and assigns the location and size to the instance when it is mapped.

### 8.4.3.3 Mapping of an Agent Instance

An agent instance may or may not be mapped together with its parent. Agent instances of the composite group attempt to map each of their children when they are mapped unless a child is specified as "not mapped with its parent". If an agent instance is specified as "not mapped with its parent", it can be mapped either by: (1) specifying that it is to be mapped when a selection occurs within another agent instance, or (2) by the application explicitly calling a GUIDES interface function to do so.

A mapped agent instance of the restricted composite group decides for each of its children individually when the child should be mapped. In other words, the children of a restricted composite agent instance do not have to be mapped together with their parent. When a composite or restricted composite agent instance is unmapped, all its children are unmapped with it.

### 8.4.3.4 Activation of an Agent Instance

A mapped agent instance is ready to accept events. It is usually placed in an active state when the cursor enters the window of the agent instance. An agent instance can also be activated by other means if such a means is defined for the particular agent class. Most agent instances do not have to do anything special when they are in an active state. However, some instances such as *Dialogue Windows* grab events when they are in an active state.

### 8.4.4 Agent Attributes

There are two types of attributes of agent instances: (1) appearance attributes, which specify the appearance of an agent instance when it is displayed on screen, and (2) behavior attributes, which specify the behavior of

an agent instance such as the I/O semantics.

To handle the agent appearance attributes with relative ease, and to provide a capability to utilize a pre-defined appearance for certain agents, the concept of style is employed. A style in general is a combination of display attributes and drawing primitives. Several standard styles are provided by GUIDES, and applications are allowed to customize these standard styles. Thus, a style may be used as an attribute for agent instances.

In GUIDES, the appearance and behavior attributes of agent instances are referred to in general as agent resources. Agent resources may be specified for an agent instance during its creation and at any time after it is created. A resource consists of two pieces of information, the resource name and the resource value. A resource name is a pre-defined name which serves as the identifier of the resource. The resource value can be any type of data, such as string, integer, or float. The type of the resource value is determined uniquely by the associated resource name. In general, if an agent does not recognize a resource that is specified for one of its instances, it will just ignore the resource.

An agent instance interprets its resources in the following order, from lowest to highest precedence:

1. General default resources for all agent classes;
2. Class default resources for the agent's class;
3. Resources specified by the parent of the agent instance, if the agent instance is created by its parent;
4. Resource assignment statements in a description file, if a way for doing so has been defined for the agent.

#### 8.4.5 Agent Semantics

An agent behaves as specified by the general rules discussed in this section as well as the rules defined by its class. This is true whether the agent is used as a standalone user-interface tool or as a component of a user-interface tool.

A mapped agent instance may have up to three display modes if the agent accepts user input. Namely, these modes are the *normal* mode, the *previewing* mode, and the *selecting* mode. An agent instance can change its mode in response to related event. However, not all agent instances of any class has all these three modes, and the events that cause an agent instance to change from a certain mode to another may not be the same for different agent classes. The relationships between agent mode changes and events depend on the semantics of agent classes, which, however, are fixed for each agent class.

Callback lists are defined for all agent instances. In general, a callback list is associated with each mode change as well as each change to a mapped or unmapped state. A callback list can contain one or more callbacks, or it can be empty. The application as well as other agent instances can register callbacks to these lists. The callback lists of an agent are pre-defined slots by which the semantics of the agent are connected to the application functionality or the semantics of other agents.

Two callback lists are common to agents of any type. These are the lists associated with mapped and unmapped state changes, and they are referred to as the mapping and unmapping lists. Callbacks in the mapping list of an agent instance will be called when the instance is mapped, and callbacks in the unmapping list will be called when it is unmapped.

There are five types of callback lists associated with agent mode changes. These callback lists and their associated mode changes are listed in Table 8.1. When the appropriate event or events occur in the window of the agent instance, the agent instance changes its mode, and the associated callbacks in

the list will be called. Both the correspondence of an event or an event sequence to a mode change and the association of a callback list with the mode change are specific to each agent class. Some agent classes maintain all the five callback lists as shown in Table 8.1 for each instance of that class. Some agent classes may not have any callback lists of this type at all. This is the case if there are no mode changes associated with agent instances of a particular class. As an example, the semantics of the *Button* agent are listed in Figure 8.5.

Table 8.1 Callback lists and agent mode changes

callback list	mode changes	
	from	to
previewing	normal	previewing
previewing_done	previewing	normal
selecting	previewing or normal	selecting
select_quit	selecting	normal
select_done	selecting	previewing

The mapped/unmapped state changes of an agent instance can be connected to the semantics of other agent instances. For example, the selection of an item in a *Static Menu* may cause a *List* to be mapped, and selection of another item at that time may unmap the *List*. This type of functionality can be accomplished by specifying the semantic connections between agent instances in the agent definitions of a description file. No C language programming is needed to do this. See Section 8.6.6 for details of this type of semantics connection.

BUTTON_UP_ENTER	. change to previewing mode from normal mode. . call previewing_callbacks if there are any
BUTTON_UP_EXIT	. change to normal mode from previewing mode . call previewing_quit_callbacks if there are any
BUTTON_DOWN_ENTER	. change to selected mode from normal mode . call selecting_callbacks if there are any
BUTTON_DOWN	. change to selected mode from previewing mode . call selecting_callbacks if there are any
BUTTON_DOWN_EXIT	. change to normal mode from selecting mode . call selecting_quit_callbacks if there are any
BUTTON_UP	. change to previewing mode from selecting mode . if the selecting_done_callback_list is not empty, call selecting_done_callbacks; if the selecting_done_callback_list is empty, put the button event information to the event register, then return GS_EXIT
BUTTON_CLICK	. change to selecting mode from previewing mode . if the selecting_done_callback_list is not empty, call selecting_done_callbacks; if the selecting_done_callback_list is empty, put the button event information to the event register, then return GS_EXIT . change to previewing mode from selecting mode
BUTTON_DOWN_STILL   BUTTON_DOWN_MOTION	. call selecting_callbacks if there are any

Figure 8.5 Semantics of the *Button* agent

#### 8.4.6 Description of Each Agent Class

This section gives a brief description of each of the agent classes developed in the present work. Details of these classes are given in (White et al, 1990).

##### 8.4.6.1 Simple Agent Group

There are five agent classes in simple agent group. These classes are derived directly from the *Basic* abstract class.

A. *Message Agent*: A *Message* agent displays one or more lines of text strings in its window and can only be displayed in the *normal* mode. *Message* agent instances do not accept any events, and therefore, they do not respond to any type of user action.

B. *Button Agent*: A *Button* agent displays a one line text string in its window. It uses three different display modes to represent its states including the *normal* mode, the *previewing* mode, and the *selected* mode. A *Button* agent instance has five callback lists associated with its mode changes: the *previewing*, *previewing\_quit*, *selecting*, *select\_quit*, and *select\_done* lists.

C. *Entry Agent*: An *Entry* agent displays a one line text string in its window, and can only be displayed in the *normal* mode. The text string in an *Entry* agent instance can be edited by the user when the *Entry* agent instance is activated. An *Entry* agent instance maintains one callback list, the *select\_done* list, and the list is associated with any change from an active to an inactive state.

D. *Toggle Agent*: A *Toggle* agent displays a one line text string in its window. A *Toggle* agent uses two different display modes to represent its states: the



*normal* mode (unhighlighted) and the *selected* mode (highlighted). A *Toggle* agent instance maintains one callback list, the *select\_done* list associated with its mode changes either from *normal* to *selected* or from *selected* to *normal*.

E. *Application Window Agent*: The *Application Window* agent is a special agent class. No I/O semantics or attributes are specified for an *Application Window* agent by GUIDES. These details depend specifically on the application, and the application is expected to handle them. The application draws whatever geometry etc. it wishes in the *Application Window* through calls to HOOPS. Callbacks associated with events in an *Application Window* can be registered to the segments in the *Application Window* through calls to the Callback Manager. Defining the *Application Window* as an agent class is for the sake of consistency of the GUIDES design and for convenience of the application.

#### 8.4.6.2 Composite Agent Group

There are two agent classes in the composite agent group, the *Form* class and the *Dialogue Window* class. These two classes are derived from the *Composite* abstract class. There is another agent class, the *Root* class, which is derived from the *Form* class. Any class which is derived from the *Composite* abstract class may have an optional label which is a *Message* agent. These classes are described below.

A. *Form Agent*: A *Form* agent instance is simply a container that can hold one or more agent instances of any other type such that these instances can be manipulated together. A *Form* agent instance can be displayed only in one mode, the *normal* mode. There are no direct I/O semantics for the *Form* agent class. Each child instance in a *Form* responds to the user's actions individually according to the child's own I/O semantics.

B. *Dialogue Window Agent*: A *Dialogue Window* agent instance can contain one or more agent instances of any agent class and can only be displayed in one mode, the *normal* mode. A *Dialogue Window* is used to accomplish a dialogue session between the user and the application. Therefore, the *Dialogue Window* is displayed on the screen only temporarily. The *Dialogue Window* is mapped to begin the dialogue session, and it is unmapped when the dialogue session terminates.

A *Dialogue Window* grabs events when it is mapped, that is, incoming events will be grabbed to the children of the *Dialogue Window* such that no event will be sent to anywhere outside the *Dialogue Window*. Thus, the user will be forced to finish the dialogue session before any other interaction can proceed. There are no direct I/O semantics for the *Dialogue Window* class, and no callbacks can be registered with a *Dialogue Window* directly. Each child instance in a *Dialogue Window* responds to the user's actions individually according to its own I/O semantics.

C. *Root Agent*: The *Root* is a special class derived from the *Form* agent class, and only one *Root* instance exists in any application. This instance is at the top of the composition hierarchy of the user-interface tools. All other agent instances in the user-interface are children and/or other descendents of the *Root*. The *Root* instance is automatically created and mapped by GUIDES at the initialization stage.

#### 8.4.6.3 Restricted Composite Agent Group

There are seven agent classes in the restricted composite agent group. These classes are all derived from the *Restricted Composite* abstract class. These classes are described below.

A. *Labeled Entry*: A *Labeled Entry* agent is simply an *Entry* agent which has a label. The I/O semantics and the application interface functions are the same for both the *Entry* and *Labeled Entry* agent classes.

B. *Notifier*: A *Notifier* is essentially a menu of *Toggle* instances and is composed of one or more *Toggle* agent instances. Each *Toggle* has the same size and is called an item of the *Notifier*. These items may be packed in the *Notifier* window either horizontally, vertically, or in the form of a two-dimensional array. A *Notifier* instance does not have any direct I/O semantics of its own. The appearance and I/O semantics of its items are the same as those of the *Toggle* agent. The callback associated with an item is called when the item changes mode. A *Notifier* instance can be classified as either a single-selection or a multiple-selection notifier. In a single-selection *Notifier*, only one of the items may be selected/highlighted at one time. When one of the items is selected, the previously selected item becomes unselected. In a multiple-selection *Notifier*, any of the items can be in the selected mode at any one time.

C. *Choice*: A *Choice* agent instance is composed of one *Button* instance. The *Button* of a *Choice* agent instance may be used to select two or more options, each of which is represented by a text string. The currently selected option is displayed in the *Button* instance. A *Choice* instance does not have any direct I/O semantics of its own. Its appearance and I/O semantics are handled by the *Button* instance. However, the *Choice* agent class specializes the behavior of the *Button*. First, the *select\_done* callback of the *Button* is made only after the cursor moves out of the *Button* instance. Secondly, any selection of the *Button* only results in the next option of the *Choice* being displayed. Finally, only one callback can be registered with each option of the *Choice*, the *select\_done* callback.

D. *Static Menu*: A *Static Menu* instance is composed of one or more *Button*

agent instances. Each of these *Buttons* has the same size and is called an item of the *Static Menu*. These items may be packed in the *Static Menu* window either horizontally, vertically, or in the form of a two-dimensional array. A *Static Menu* instance does not have any direct I/O semantics of its own. The appearance and I/O semantics of its items are the same as the *Button* agent.

E. *Popup Menu*: A *Popup Menu* instance is composed of one or more *Button* agent instances. Each of the *Buttons* has the same size and is called an item of the *Popup Menu*. The items may be packed in the *Popup Menu* window either horizontally, vertically, or in the form of a two-dimensional array. The appearance and I/O semantics of its items are the same as the *Button* agent. A *Popup Menu* agent instance is displayed on screen only temporarily. In general, a *Popup Menu* can be mapped by a certain event in an application window, by the selection of another agent instance, or by calling a GUIDES interface function.

When a *Popup Menu* is mapped, all the incoming events will be grabbed to the *Popup Menu*, and all of the *Popup Menu*'s items will be displayed. If the mouse cursor is moved into an item with one of the mouse buttons down, the item entered will be highlighted in the *selected* mode and the *selecting* callback associated with this item, if there is one, will be called. If the mouse button is released or clicked while on an item, the *select\_done* callback, if there is one, will be called, and the menu will be unmapped. If the mouse button is released or clicked while the mouse cursor is outside the menu area, the menu will be unmapped without the *select\_done* callback being made.

F. *List Column*: A *List Column* instance can only be defined and used within a *List* agent and can not be used as an independent user-interface tool. A *List Column* is composed of a number of children agent instances referred to as elements. It maintains a group of data, each item of which is called an item of

the *List Column*. One item is displayed in an element, and the number of items may not be the same as the number of elements. If the number of elements is greater than the number of items, the extra elements will be unmapped. If the number of items is greater than number of elements, only some of the items are displayed. The items in this case can be scrolled in the *List Column*.

Generally, elements of a column are packed vertically in the column, as suggested by the name. However, the elements in a column can also be packed in multiple columns, which may be convenient if only one group of data items is displayed in a *List*.

Items displayed in a column can be classified in five types: *fixed*, *number*, *highlightable*, *selectable*, and *changeable* types. Items displayed in the same column must be of the same type. In general, the content of the items can be any one of four data types: String, int, float, or a pair of floating point numbers. Different types of agents may be used to compose a *List Column* instance according to the type of its items.

A *select\_done* callback may be registered with a *List Column* agent instance, and the callback is called whenever one of its element changes states.

**G. List:** A *List* agent instance is composed of one or more *List Column* agent instances and is used to display multiple groups of data on screen. Each group of data is displayed in a *List Column* instance. There is no limit on the number of data items in each group. However, the number of data items and the number of elements must be the same for each *List Column*. The number of data items in the columns of a *List* can be dynamically specified by calling an interface function of the *List* agent.

The main distinctive features of the *List* agent are as follows:

1. The number of items contained in the columns of a *List* instance may be modified at run time, and the contents of these items are specified at run time.
2. In general, only a certain number of items in each *List Column* are displayed on the screen at one time. The items in each *List Column* can be scrolled such that the desired items will be visible in the display.

If the number of items contained in each column is larger than the specified number of elements that can be displayed on the screen, the *List* may use two *Button* agent instances for scrolling of the items displayed. One of the two *Buttons* is for scrolling the items one line at a time, and the other *Button* for scrolling the items one page at a time. A page is defined as the number of elements being displayed on the screen.

A *List* agent can be registered with a single *select\_done* callback which will serve as the default callback for each of its columns. This default callback can be overwritten for an individual column by defining a specific *select\_done* callback for the column.

## 8.5 Graphical Utilities

The graphical utilities of GUIDES are similar to agents in that they are general interaction techniques and can be utilized for any application. Typical graphical utilities are the Keypad, and the X-Y Plot Manager. However, there are some major differences between agents and graphical utilities:

1. Agents are more general, and can be used for any purpose, while graphical utilities are designed to accomplish pre-defined specific but application-independent tasks. For example, the X-Y Plot Manager is used to display x-y curves in an application window. It contains a number of its own non-graphical functions for processing the x-y curves.

2. Agents can be composed to build complex interaction techniques, while graphical utilities are pre-defined and fixed, and cannot be further composed. For example, the *Button* agent can be used for any purpose. However, the X-Y Plot Manager can only be used to display x-y curves.
3. The layout within the window of a graphical utility is usually pre-defined and need not be specified by application programmers in their description files.
4. Agents never have access to the application database, and the only links between agent instances and application code are callbacks. Graphical utilities may need to access to the application database in order to provide fast feedback. For example, the X-Y Plot Manager needs access to the curve data in the application database.

The X-Y Plot Manager utility has been developed in this work. Other graphical utilities may be developed in the future work.

## 8.6 The Description Language

GUIDES provides a Description Language for applications to present the specification of their user interfaces. The files containing user-interface specifications written in the description language are called description files. The Parser of the description language parses the description files passed to GUIDES at run-time, and, by communication with GUIDES agents, builds the user-interface.

This section describes briefly the features of the description language and the use of the language in defining a user-interface. The syntax and features of the language are presented in Sections 8.6.1 to 8.6.8. A complete example of an application is given in Section 8.6.9 to illustrate these features. A more detailed explanation of the language can be found in the GUIDES Reference Manual

(White et al., 1990), and the implementation of the description language parser is reported elsewhere (Zhang et al., 1990).

### 8.6.1 Lexical Conventions

There are seven token types defined in the GUIDES description language: comments, identifiers, keywords, constants, strings, operators, and separators.

1. **Comments:** A comment line is delimited either by a pair of characters `"/*` and `*/` or by the double dash (`--`). Comments may not be nested and may not be continued for more than one line.
2. **Identifiers:** An identifier is a pre-defined resource name. These resource names always start with the characters `"GsR"` and denoted in general by the name *GsRresource*.
3. **Keywords:** There are 15 names reserved for use as keywords and may not be redefined in the specification of the user interface.
4. **Constants:** Constants may be of one of the following types: integer, floating point number, logical (true and false), a pair of floating point numbers, text string, and predefined constant. Any constant of these types listed in the following are denoted as either *CONSTANT* or its type name in all capital letters, e.g., *INTEGER* for an integer constant.
5. **Strings:** A string is a sequence of characters surrounded by double quotes. A string is denoted as *STRING* in the description of the language. A string can also be used as a constant.
6. **Operators:** There is only one operator, the colon, `':`, in the language, which is used in resource assignment statements to separate the name and the value of a resource.



7. Separators: Either the new-line character or the semi-colon character ';' can be used to separate statements, and both are denoted as NL. The open brace '{' and the closed brace '}' are used to delimit a compound statement.

### 8.6.2 Statements and Compound Statements

Typically, most of the statements in a GUIDES description file are resource assignment statements and have the form

*GsResource* : *CONSTANT*

The appropriate data type of the *CONSTANT* depends on the type of the resource being defined by the statement. Several statements can be grouped together to form a compound statement. A compound statement is expressed as

{ *statement-list* }

where the *statement-list* represents one or more statements, and can be empty. A compound statement has no meaning by itself. It is always associated with other statements which require its use.

### 8.6.3 Defining an Agent Instance

The statement

*agent* *AGENT\_CLASS* *STRING* { *statement-list* }

defines an agent instance. *AGENT\_CLASS* is a pre-defined constant and is the class identifier of the agent instance to be defined. *STRING* is the name of the agent instance. The statements which form the compound statement specify the details of the agent instance. An agent may be defined inside the compound

statement of the definition of another agent. An agent so defined can only be referred to in the compound statement. This enables the local definition of agent instances and makes nested agent definitions possible.

Default resources for a certain agent class may be specified by the set default statement as below

*set AGENT\_CLASS defaults { statement-list }*

where *AGENT\_CLASS* is the identifier of the class to be set, and the compound statement contains the default resources to be used by instances of that class.

With the exception of the *Root* instance, an agent instance can only be defined once. The *Root* instance is created by GUIDES automatically at initialization time. Agent definition statements for the *Root* agent may be entered more than once for an application to instance its child agents and to set additional resources.

#### 8.6.4 Defining a Composite Agent Instance

A unrestricted composite agent instance is composed by instancing agent instances (its child agents) which have already defined elsewhere. It instances a child agent by the following statement

*instance AGENT\_CLASS STRING PAIRDATA PAIRDATA*

inside the compound statement of its definition. *AGENT\_CLASS* and *STRING* are the class name and name of the child agent to be instanced. The location and size of the child agent within the window of the composite agent instance being defined are specified by the first and the second *PAIRDATA* respectively.

### 8.6.5 Defining a Restricted Agent Instance

The child agents of a restricted composite agent instance are usually referred to as its items. These child agents are specified in the compound statement in the definition of the restricted composite agent by the item statement as below

*item STRING { statement-list }*

where *STRING* is the name of the item, and compound statement may include any necessary statement to specify the item.

### 8.6.6 Connecting the Agent Semantics

The semantics of some agent classes can be connected such that the selection of an agent instance will cause another instance to be mapped. For example, a *Dialogue Window* can be specified to be mapped by the selection of a *Button*. The following *mapped\_by* statement establishes such a connection

*mapped\_by AGENT\_CLASS STRING*

or

*mapped\_by AGENT\_CLASS STRING STRING*

The first case specifies the class identifier and the name of the agent instance by which the instance being defined will be mapped. The second case specifies the class identifier, the name of the agent instance, and name of the item by which the instance being defined will be mapped.

Similar connections may also be established by the *unmapped\_by* statement such that the selection of an agent instance will cause another instance to be unmapped. The *unmapped\_by* statement uses the keyword *unmapped\_by* and has the same form as the *mapped\_by* statement.

### 8.6.7 Defining an Agent Style

The appearance of an agent instances is specified by the style used by the instance. GUIDES provides a default style for each agent class. An application may change the default style of an agent class. It can also define new styles based on existing styles. This is achieved by the style definition statement

```
define style STRING { statement-list }
```

where *STRING* is the name of the style being defined. The compound statement specifies the details of the style.

### 8.6.8 Other Features

To increase the readability of a description file, aliases may be defined and used to replace identifiers, constants, or even other aliases.

To build a large scale user-interface, the number of agent instances used may be quite high and the specification may be long. The description language provides the file inclusion statement so that the specification can be separated into several files. These files can then be included in a "main" file which is passed to GUIDES.

Any file which specifies a portion of the complete user-interface of an application may be processed by GUIDES to create this portion of the interface during any stage after the initial startup of the application. Thus, it is possible to defer the creation of an off-screen portion of an interface until this portion needs to be displayed. This feature can be useful to improve the start-up performance of an application.

### 8.6.9 A Complete Example

Figure 8.6 shows a modified version of the *Goodbye world* application implemented using the GUIDES. This application is discussed previously in Chapter 6 in regard to use of the X11 Toolkit. Figures 8.7 and 8.8 list respectively the C code, "goodbye.c", and the description file, "goodbye.r", of this application. It should be noted that for this application, the description file can be created much smaller than the one shown in Fig. 8.8. Many statements are not necessary for this application and included in Fig. 8.8 to show features of the GUIDES description language.

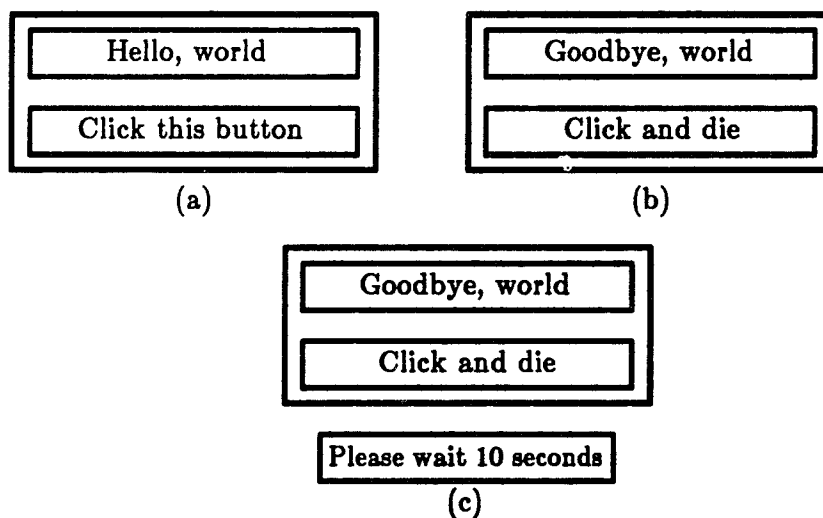


Figure 8.6 A modified version of the *Goodbye world* application

In the description file, text preceded by double hyphens (--) is a comment. Two *Form* agent instances, "f1" and "f2", and a *Message* agent instance, "wait", are defined and instanced by the *Root* agent instance. Each *Form* instance contains two children, one *Message* instance and one *Button* instance. In the *Form* "f1", the *Message* "m1" displays the text string "Hello, world", and the *Button* "b1" displays "Click this button". In the *Form* "f2", the *Message* "m2"

```
#include "guides.h"

int
quitcallback(client_data, event_info)
Gs_ClientData client_data;
Gs_EventInfo event_info;
{
    GS_MapMessage("wait");
    GS_UpdateDisplay();
    sleep((int) client_data);
    return GS_EXIT;
}

int
main()
{
    static Gs_CallbackRec sbCallbackTable[] = {
        {"quit", quitcallback, (Gs_ClientData) 10 },
    };
    auto Gs_EventRec event;

    GS_Initialise();
    GS_DefineCallbackTable(sbCallbackTable, 1);
    GS_ParseDescriptionFile("goodbye.r");
    GS_EventManager(&event);
    exit(0);
}
```

Figure 8.7 The C code of the *Goodbye world* application

```

define call GsRselect_doneCallback      -- define the name call as an alias
define style "user_defined_style" { -- define a style
  GsRstyle : "2d_appearance"      -- load a GUIDES supplied style
  GsRmode : normal                 -- change the normal mode
  GsRbackgroundColor : "yellow"   -- set the background color of the normal mode
}
set form defaults {                -- set the default attributes of form
  GsRstyle : "user_defined_style"
}
agent message "wait" {             -- define a message
  GsRstring : "Please wait 10 seconds"
  GsRmappedWithParent : false      -- not mapped with parent
  GsRstringAlignment : left
  GsRstringSize : 0.80
  GsRstyle : "user_defined_style"
}
agent root "root" {               -- the Root agent
  GsRlocation : (-1.0, -1.0)
  GsRsize : ( 2.0, 2.0)
  agent form "f1" {               -- the form on the first page
    agent message "m1" { GsRstring : "Hello, world"
                        GsRstringAlignment : center }
    agent button "b1" { GsRstring : "Click this button" }
    instance message "m1" (-0.8, 0.1) (1.6, 0.7)
    instance button "b1" (-0.8, -0.8) (1.6, 0.7)
    unmapped_by button "b1" -- unmap when the button b1 is selected
    GsRstyle : "2d_appearance" -- set the style to overrides the class default style
  }
  agent form "f2" {               -- the form on the second page
    agent message "m2" { GsRstring : "Goodbye, world"
                        GsRstringAlignment : center }
    agent button "b2" { GsRstring : "Click and die"
                        call : "quit" }
    instance message "m2" (-0.8, 0.1) (1.6, 0.7)
    instance button "b2" (-0.8, -0.8) (1.6, 0.7)
    GsRmappedWithParent : false
    mapped_by form "f1" "b1" -- map when the button b1 in f1 is selected
  }
  instance form "f1" (-0.8, -0.5) (1.6, 1.0)
  instance form "f2" (-0.8, -0.5) (1.6, 1.0)
  instance message "wait" (-0.4, -0.9) (0.8, 0.1)
}

```

Figure 8.8 The description file of the *Goodbye world* application

displays "Goodbye, world", and the *Button* "b2" displays "Click and die". The *Message* instance "wait" displays "Please wait 10 seconds".

When the program starts, only the *Form* "f1" will be displayed on the screen as shown in Fig. 8.6(a) because the *Form* "f2" and the *Message* "wait" are defined as not being mapped with their parent, the *Root*. The *Form* "f1" is also defined as unmapped by the *Button* "b1", and "f2" is defined as mapped by the *Button* "b1" in "f1". Thus, when "b1" is selected by clicking a mouse button while the mouse cursor is in the window of "b1", the *Form* "f1" as well as its children will disappear, and the *Form* "f2" and its children will be displayed. This is shown in Fig 8.6(b). A callback with the name "quit" is registered to the *Button* "b2". This callback is called by GUIDES when the "b2" is selected.

The callback function, *quitcallback*, associated with the callback name "quit" is listed in Figure 8.7. When this function is called, it first calls a GUIDES interface function *GS\_MapMessage* to map the *Message* instance "wait" with the name of the *Message* instance as a parameter. It then calls *GS\_UpdateDisplay* to update the screen or else the screen will not be updated until the next event is processed. The resulting display on the screen is shown in Figure 8.6(c). The callback function also calls a system function *sleep* to pause the process for a few seconds specified by the client-data, and then returns *GS\_EXIT*. Therefore, once this callback function is called, the interaction control, which was passed to GUIDES from the application by calling the interface function *GS\_EventManager*, will be returned to the application.

Also shown in Figure 8.7 is the *main* function of the application. In the *main* function, the callback function *quitcallback* is associated in the array of *sbCallbackTable* with a callback name "quit" and the client-data, an integer 10 which is the number of seconds the application should pause when the button "b2" is selected. In the *main* function, the application first calls *GS\_Initialize* to initialize GUIDES. It then calls *GS\_DefineCallbackTable* to register callbacks to



GUIDES and *GS\_ParseDescriptionFile* to tell GUIDES to parse the description file "goodbye.r". Finally, the application passes the interaction control to GUIDES by calling the function *GS\_EventManager*. The execution of the application pauses for 10 seconds and then terminates when a mouse button is clicked while the mouse cursor is in the *Button* "b2". This causes GUIDES to return control to the application since the callback function *quitcallback* returns *GS\_EXIT* to GUIDES.

It should be noted that the *Message* "wait" could be specified as *mapped\_by* the *Button* "b2" in the *Form* "f2". The reason for not doing so is to show more features of the GUIDES. The "goodbye.r" file also shows other features of GUIDES description language. A name "call" is defined as an alias of the resource name *GsRselect\_doneCallback* and is used in specifying the *select\_done* callback of the *Button* "b2". A style named "user\_defined\_style" is defined by a style definition statement. This style is the same as a GUIDES provided style "2d\_appearance" except the background color of the *normal* mode is changed to "yellow". This style is specified as the class default style of the *Form* agent class in a set class default statement. It is used by the *Form* "f2" because this style is the class default for *Form* agent, and by the *Message* "wait" because this style is specified explicitly by a resource assignment statement, *GsRstyle:"user\_defined\_style"*, in the definition of the *Message* "wait". This style is not used by the *Form* "f1" because a resource assignment statement, *GsRstyle:"2d\_appearance"*, in the definition of "f2" overrides the class default.

The benefit of using GUIDES is obvious from this example. We are able to implement the user-application interaction entirely in the GUIDES description language. The static layout and appearance of the interface is independent of the C code of the application. Moreover, the run-time semantics associated with mapping and unmapping of GUIDES agents are also independent of the C code. Thus, GUIDES makes it possible to develop and test the user-interface independently from the application-specific components.

Often, the user-interface can be developed and tested before the application-specific code is written.

## 8.7 GUIDES versus Motif

The OSF/Motif system is described in Chapter 6. It is interesting to compare the use of Motif User Interface Language in defining application user interfaces with the use of the GUIDES description language. Both systems are similar conceptually in that they are all developed from the motivation to separate the definition of the user interface from the specific functionalities of the application. In some aspects, Motif provides more functionality than GUIDES for building application's user-interfaces.

The similarities and differences between GUIDES and Motif are listed below.

1. Both systems use a description (or specification) language to describe the interface of an application separately from the application's C code and to create the interface at run time.
2. Motif provides more interface tools for applications than GUIDES. However, GUIDES provides a reasonably complete set of tools for application software development. New interface tools (i.e., new widget classes) may be defined in Motif UIL, while this can not be done in GUIDES description language.
3. Motif provides a compiler to compile the user interface specification, and the compiled form of the specification is loaded at run time. This leads to a higher run-time performance in the creation of the interface. GUIDES does not have a compiler for its description language.

4. Both GUIDES and Motif allows applications to defer the creation of certain interaction tools that are initially off-screen until these tools need to be displayed. This helps to improve the start-up performance.
5. GUIDES works with a modern three-dimensional graphics package HOOPS seamlessly. It is not known whether there is a corresponding three-dimensional graphics package that works seamlessly with Motif.
6. Both GUIDES and Motif use the callback mechanism for communication between the user-interface and application specific components. However, in Motif the binding of a callback function with the client-data is done in the specification language, whereas in GUIDES it is done in the application's C code. Motif's approach is flexible in that the client-data may be specified independently of the C code. However, this flexibility complicates the usage of the language. GUIDES's approach is less-flexible but is still reasonable because the client-data is usually data defined in the application's C code and should be naturally bound with the callback functions in the C code. Also, the GUIDES approach simplifies the reference to callbacks in the description language.
7. The Motif UIL can be used to specify only the static appearance of the user-interface. The GUIDES description language can be used to specify not only the static appearance, but also the dynamic behavior of the interface, i.e., the mapping and unmapping of interaction tools. This is convenient for the design of user-interfaces composed of several different screen setups.
8. The Motif UIL provides instructions for string concatenation, for defining literal values used in a specification and fetched by the application C code, for defining identifiers defined in application C code and used in the specification. These features are not supported in GUIDES description language. However, they may be easily added to the language.

## LIST OF REFERENCES

1. Brown, M. D., "Understanding PHIGS", TEMPLATE, The Software Division of Megatek Corporation, San Diego, California, 1985.
2. Cox, B., and Hunt, B., "Objects, Icons, and Software-ICs", in Tutorial: Object-Oriented Computing, Vol.2, Implementations, Edited by Gerald E. Peterson, Computer Society Press, 1987, pp.99-108.
3. Dodani, M. H., Hughes, C. E., and Moshell, J. M., "Separation of Powers", BYTE, March 1989, pp.255-262.
4. Fiedler, D., "Future Imperfect", BYTE, May 1989, pp.227-234.
5. Greenberg, R. M., "Faces of Unix", PC Magazine, September 1989, pp.143-157.
6. Hartson, R., "User-Interface Management Control and Communication", IEEE Software, January 1989, pp.62-70.
7. Hayes, F., Baran, N., "A Guide to GUIs", BYTE, July 1989, pp.250-257.
8. Hurley, W. D. and Sibert, J. L., "Modeling User Interface-Application Interactions", IEEE Software, January 1989, pp.71-77.
9. Ingraffea, T. and Mink, K., "Project SOCRATES: Fostering a New Collegiality", Academic Computing, October 1988, pp.20-21, 60-63.
10. Kliever, B. D., "Powerful Portable 3-D Graphics", BYTE, July 1989, pp.193-198.
11. Langa, F., "Open Everything", BYTE, April 1989, pp.6.
12. McCormack, J., Asente, P., and Swick, R. R., "X Toolkit Intrinsic - C Language X Interface", Technical Report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988a.
13. McCormack, J., and Asente, P., "Using the X Toolkit or How to Write a Widget", submitted to USENIX, 1988b.
14. Mednieks, Z. R., and Schilke, T. M., "C Programming Techniques for the Macintosh", Howard W. Sams & Co., 1986, 322pp.
15. Myers, B. A., "User-Interface Tools: Introduction and Survey", IEEE Software, January 1989, pp.15-23.

16. Open Software Foundation, "OSF/Motif Programmer's Guide", Prentice Hall, New Jersey, 1990.
17. Paul, J., "OSF/Motif", BYTE, May 1989, pp.230-231.
18. Schmucker, K., "Object-oriented Programming for the Macintosh", Hayden Book, 1986.
19. Schmucker, K., "MacApp: An Application Framework", in Tutorial: Object-Oriented Computing, Vol.2, Implementations, Edited by Gerald E. Peterson, Computer Society Press, 1987, pp.72-91.
20. Seymour, J., "The GUI An Interface You Won't Outgrow", PC Magazine, September 1989, pp.97-109.
21. Shell, B., "Running HyperCard with HyperTalk", Management Information Source, 1988.
22. Swick, R. R., and Weissman, T., "X Toolkit Widgets - C Language X Interface", Technical Report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988.
23. Thompson, T., "The Next Step", BYTE, March 1989, pp.265-269.
24. Weiskamp, K., and Shammas, N., "Mastering HyperTalk", John Wiley & Sons, 1988.
25. White, D. W., Zhang, H., Tan, L., and Chen, W. F., "GUIDES Reference Manual", Structural Engineering Report, CE-STR-90-8, School of Civil Engineering, Purdue University, Indiana, June 1990.
26. Wiegand, G., "HOOPS Reference Manual (Second Edition)", Ithaca Software, Ithaca, New York, 1988.
27. Zhang, H. and Chen, W. F., "GRAFIC/CE88 - An Interactive Graphics Package", Structural Engineering Report, CE-STR-88-19, School of Civil Engineering, Purdue Univ., West Lafayette, Indiana, July, 1988a, 102pp.
28. Zhang, H. and Chen, W. F., "User's Manual of LineGraph", Structural Engineering Report, CE-STR-88-21, School of Civil Engineering, Purdue Univ., West Lafayette, Indiana, 1988b, 108pp.
29. Zhang, H., Tan, L., White, D.W., and Chen, W.F., "Design and Implementation of GUIDES: A Graphical User-Interface Development System", Structural Engineering Report, CE-STR-90-7, School of Civil Engineering, Purdue Univ., West Lafayette, Indiana, June, 1990, 112pp.

**PART THREE**

**OBJECT CLASSES FOR ENGINEERING COMPUTING**

## CHAPTER 9 THE SESDE OBJECT LIBRARY

This chapter gives a brief summary of state-of-the-art approaches in engineering software development and introduces the SESDE object library. Section 9.1 reviews several recent literatures and discusses some issues in object-oriented engineering software development. Section 9.2 introduces the SESDE object library. This library currently includes a set of object classes for engineering computing in general. The design and implementation of selected classes of the library are discussed in Chapters 10, 11, and 12. A complete documentation of the library may be found in (Zhang et al., 1990b).

### 9.1 Object-Oriented Engineering Software Development

#### 9.1.1 Literature Review

The term engineering software can refer to software for different activities including engineering analysis, design, research and education. Engineering software is typically associated with a certain engineering domain and implements some physical principles and related mathematical formulations and algorithms in that specific engineering domain. Therefore, expertise in a specific engineering domain is necessary to the development of quality engineering software. This is especially true for research software developed for testing new models or new methods. Thus, engineers or engineering professionals must play the critical role in the development of engineering software rather than computer science professionals. This is the major distinction between engineering software and software such as operating systems. This is also a reason why the adoption of new software engineering techniques is occurring at

a slower pace in engineering software development.

Recently, object-oriented programming has attracted considerable attention in engineering software development (Miller, 1988; Baugh et al, 1989; Fenves, 1990). Problems with the older software development approach in procedural languages such as low levels of abstraction, over-complicated and expensive in development and maintenance have been recognized. Attempts have been made to apply object-oriented programming in engineering software development.

Miller (1988) developed an object-oriented structural analysis program for linearly connected finite element systems (i.e., systems in which no one node connects to more than two elements). The program was implemented in Flavors, an object-oriented extension of the LISP language. Miller also discussed an object-oriented modeling for general structural systems.

Baugh and Rehak (1989) have provided an excellent description of object-oriented design of finite element programs for structural analysis. Their paper focused on abstracted data representations of entities in a finite element system. A prototype finite element system is reported as being developed in their work to demonstrate this new approach. However, no details are given regarding the design and implementation techniques, the efficiency of their implementation, and the implementation language.

Fenves (1990) discussed the advantages of object-oriented programming for developing engineering software. In his paper, the advantages are illustrated by the development of an abstraction of mathematical graphs used for ordering nodes and elements in a finite element mesh. The Smalltalk language is used in the development. Even though reusability is recognized as an important feature of object-oriented programming in this paper, the illustration example presented seems to be a one-of-a-kind design.



A programming environment for structural engineering computations was recently reported by Landers (1990). This environment consists of four major components:

1. A command interpreter which defines a language for developing applications in the environment. The syntax of the language is a mixture of both C and FORTRAN languages. The interpreter translates programs written in the language into instructions executable by a virtual machine.
2. A virtual machine which is equivalent to a complete computer system implemented entirely in software. This virtual machine is the core of the environment.
3. A set of structural analysis subroutine libraries which are callable from applications written in the language of the system.
4. A set of application libraries which consist of a collection of utilities that are used to connect applications developed in the system with existing finite element programs and computer graphics systems.

Landers claims that this system offers many advantages over the current state of technology in the area of program development for structural analysis and is highly flexible and portable. However, the following weaknesses of this system are apparent from Landers's report.

1. The language is not well designed from the point-of-view of software engineering. Global data areas or variables are defined by the environment and shared by modules or subroutines provided by the environment and used to compose an application. Thus, to program in this environment, a programmer has to learn not only the syntax of the language, but also the global variables and assumptions made on these variables to compose the modules into an application.

2. An application developed in the environment is portable only in the sense that the environment is ported together with the application.
3. The structural libraries of the environment are subroutine libraries, and subject to the criticisms discussed previously in Chapter 3. The modules contained in these libraries are reusable only in the sense of "use-as-is".
4. The language provided by the environment is an interpretive language, and thus it is likely to have low run time efficiency.

As discussed in previous chapters, object-oriented programming with object-oriented languages provides the most promising approach at the present time for developing quality engineering software in an efficient manner. There are two important issues which need to be emphasized regarding this approach: reusability and efficiency. These are discussed in the following two sub-sections.

### 9.1.2 Reusability

As discussed in Chapters 3 and 4, reusability is the most important benefit of adopting object-oriented programming. As a result of reusability, the efforts spent on engineering software development can be accumulated, rather than wasted, through the use of reusable software components. A substantial amount of engineering software can be developed by composing existing components with new components developed for the software. These new components will be added to the collection of reusable components for future reuse. Thus, the development of high-quality engineering software will be made easier. To facilitate large-scale reuse, a large collection of object classes is necessary. The class libraries for a specific engineering domain in the future may contain several hundreds even thousands classes for reuse. Efforts must be made toward the development of such libraries.

However, object-oriented programming methodologies and object-oriented languages only make the creation of reusable components feasible and much easier compared with the older methodologies and procedural languages. They do not make reuse happen. Designing a software in terms of objects and implementing the software in an object-oriented language do not imply that the components of the software can easily be reused, and one-of-a-kind applications may still be easier to develop. With object-oriented technologies, the creation of reusable components is still a challenging task. It requires a great deal of knowledge of the application domain and of object-oriented programming techniques, and well thought out design and implementation decisions.

One way to create reusable components or object classes for a specific domain is to perform a domain analysis and to actively search, classify and create object classes often used by applications in the domain. These classes usually form the foundation for reuse in the domain and for further development of other object classes.

Another way is to create reusable components in the development of a specific software. Inevitably, new classes are found necessary for the specific software and not available in class libraries. These new classes should not be designed and implemented only for the particular software, rather, they should be designed and implemented as general purpose reusable components.

### 9.1.3 Efficiency

Due to the rapid increase of computer processing speed, the efficiency of a certain language for numerical computations becomes less important now than ever before. However, numerical efficiency is still a primary concern for computationally-intensive applications such as finite element analysis programs.

Object-oriented programming performed in an object-oriented language adds overhead at run-time compared to traditional procedural programming. This overhead varies among different object-oriented languages. Considering the advantages in program design, development, maintenance, and evolution, lower efficiency of object-oriented languages may be compromised for many engineering software systems. However, for computationally-intensive applications, object-oriented languages with higher efficiency such as C++ (of which efficiency is a primary goal) should be used. Languages such as Smalltalk, which is claimed to be much purer than C++ in the sense of "object-orientedness", may be used to produce prototypes of computational-intensive systems but not for the systems for practical use because of their low run time performance.

## 9.2 Object Classes in the SESDE Library

A set of object classes for engineering computing in general is developed in the present work. These classes are written in the C++ language and stored as reusable components in the SESDE Object Library. These classes represent and implement basic entities and utilities commonly used in engineering computing. These classes can be subdivided into three groups: object classes for general data structures and general utilities, object classes for full matrices, and object classes for sparse matrices. Table 9.1 shows a complete listing of these classes.

The goals of this development are as follows:

1. To investigate and demonstrate object-oriented approaches for engineering software development;
2. To provide models for the development of other reusable components for engineering computing;

Table 9.1 Object classes developed in the present work

class name	description
<i>String</i>	for text strings.
<i>ErrorHandler</i>	error handling utility.
<i>ClArgument</i>	command-line argument processing utility.
<i>Vector</i>	parameterised vector class.
<i>ExtArray</i>	parameterised array class for collections of objects.
<i>Bag</i>	parameterised array class for collections of objects.
<i>Matrix</i>	for general full matrices.
<i>LUMatrix</i>	for decomposed matrices.
<i>DMatrix</i>	for diagonal matrices.
<i>SMatrix</i>	for symmetric matrices.
<i>LTMatrix</i>	for lower-triangle matrices.
<i>UTMatrix</i>	for upper-triangle matrices.
<i>Sparse</i>	abstract class for node-based sparse matrix classes.
<i>SVector</i>	vector class to be used with the Sparse class.
<i>ActiveColumn</i>	node-based sparse class implementing the Skyline scheme.
<i>SGraph</i>	node-based sparse class implementing a graph-based scheme.
<i>SparseMatrix</i>	abstract class for unknown-based sparse matrix classes.
<i>Skyline</i>	unknown-based sparse class implementing the Skyline scheme.

3. To provide basic reusable components for engineering software development.

Some of these classes are mainly for numerical computing such as the full and sparse matrix classes. The basic type of floating-point numbers for these classes is defined as *Real* which can be either *float* or *double*. The selection of the basic type can be made at compilation time.

## CHAPTER 10 BASIC DATA STRUCTURES AND UTILITIES

This chapter describes selected object classes for basic data structures and for general utilities developed in the present work. Section 10.1 presents the *ErrorHandler* class for exception handling. Two parameterized extensible array classes are presented in Section 10.2. Finally, Section 10.3 describes a parameterized vector class. In each section, the reason for the development is discussed first, the description of the design and use of the class(es) then follows. A complete documentation of these classes may be found in (Zhang et al., 1990b).

### 10.1 An Exception Handling Class

When object libraries are used extensively, the mechanisms for error or exception handling become important. In an application program, exception handling is relatively easy because application programmers can decide what the software should do. When a certain exception occurs, programmers can decide whether to resume the execution at the point where the exception has occurred, to resume the execution at a point other than the exception point, or simply to abort the execution.

However, in the design of a library, final decisions on the handling of exceptions should not be made by library developers. Take the matrix calculator (which will be discussed in the next chapter) as an example. If a matrix element selection function is called with bad indices, an error message should be reported and the program should then process the next command.

However, for other applications, the execution may have to be aborted for the same error in order to let programmers determine the reason for the exception through the examination of the core file with the help of debugger tools.

An object class *ErrorHandler* is developed and used by many of the object classes developed in the present work to handle exceptions in a flexible way. This class is designed to create objects used by other classes as member objects. The declaration of the *ErrorHandler* class is listed in Figure 10.1. As an example, the use of the class with the *Matrix* class described in the next chapter is shown in Figure 10.2.

The exceptions that may be detected by a class are identified by exception identifiers. In the *Matrix* class example shown in Figure 10.2, *mtr\_index*, *mtr\_size*, ... are exception identifiers. Exceptions are further classified as either error-exceptions or warning-exceptions. Whether a specific exception should be defined as an error or a warning is up to the class developer. Usually, a serious exception is considered as an error, and a less serious one, which the class developer may know how to handle, and in which the execution may be resumed is considered as a warning. This classification is to provide the clients with two different levels for exception handling. However, the *ErrorHandler* treats the two types of exceptions in a very similar way.

#### 10.1.1 Handling Error Exceptions

Once an exception occurs, the function *ErrorHandler::error* is called through an *ErrorHandler* object (*Matrix::eh* in Fig. 10.2). Here, the notation *Matrix::eh* and *ErrorHandler::error* is interpreted as *class-name::name* and indicates that *name* is the name of a member of the class *class-name*. The function *error* accepts a variable number of parameters, with the first parameter being an exception identifier, the second being a format string that specifies how to print the rest of the arguments with the message header of the class.



```
typedef int (*Ehf)(int);
const int ignore = 0, throw = 1;

class ErrorHandler {
    int    error_signal;
    int    warning_signal;
    char*  message_header;
    Ehf    error_handler;
    Ehf    warning_handler;
public:
    ErrorHandler(char* mh, int es = SIGABRT, int ws = SIGUSR1)
    { // SIGABRT and SIGUSR1 are constants defined in standard header signal.h
        error_signal = es;
        warning_signal = ws;
        message_header = mh;
        error_handler = warning_handler = (Ehf) 0;
    }
    void set_error_signal(int es) { error_signal = es; }
    void set_warning_signal(int ws) { warning_signal = ws; }
    void set_error_handler(Ehf f) { error_handler = f; }
    void set_warning_handler(Ehf f) { warning_handler = f; }
    void error(int ...);
    void warning(int ...);
};
```

Figure 10.1 The declaration of the *ErrorHandler* class

**An excerpt from the Matrix class declaration**

```

#include "errorCC.h"

enum matrixerrors { mtr_index, mtr_size, mtr_dim, .... };

class Matrix {
private:
    ....
    static ErrorHandler eh;
public:
    ....
    void set_error_signal(int i)      { eh.set_error_signal(i);      }
    void set_warning_signal(int i)    { eh.set_warning_signal(i);  }
    void set_error_handler(Ehf f)     { eh.set_error_handler(f);     }
    void set_warning_handler(Ehf f)   { eh.set_warning_handler(f);   }
    ....
};

```

**An excerpt from the Matrix class implementation**

```

ErrorHandler Matrix::eh("Matrix Error:");
Real& Matrix::operator()(int i, int j)
{
    if (i < 0 || i >= row || j < 0 || j >= col)

        eh.error(mtr_index, "Matrix index (%d, %d) out of range", i, j);
    return a[i][j];
}

```

Figure 10.2 Use of the *ErrorHandler* in the *Matrix* class

A client of a class containing an *ErrorHandler* member object for exception handling may specify an *error-handler* for the class. An *error-handler* is of type *Ehf* or a function which accepts an *int* argument and returns an *int*. In the *Matrix* class example above, the function *Matrix::set\_error\_handler* is for this purpose, which in turn calls *eh.set\_error\_handler* to specify the *error-handler* for the *eh* member object. An *error-handler* accepts an exception identifier as its argument. It returns *ignore*, a *const int* defined in *errorCC.h*, if the exception is handled successfully and the execution can be resumed at the point where the exception has occurred. Otherwise, it returns *throw*, a *const int* defined in *errorCC.h*.

In the function *error*, after the message is printed, the *error-handler* will be called if there is one. If no *error-handler* is specified or the *error-handler* returns *throw*, a signal is sent to the operating system. The signal sent by the function *error* is by default *SIGABRT*, a signal constant defined in the C standard header file *<signal.h>*. This signal may be caught by the client using the signal facilities provided in the C language. If no signal is caught by an application, sending the signal *SIGABRT* to the operating system will abort the process and generate a core file. The clients of *ErrorHandler* may reset the *error-signal* to be sent. In the *Matrix* class example, the function *Matrix::set\_error\_signal* is for this purpose. This function in turn calls *eh.set\_error\_signal* to reset the error signal of *eh*.

### 10.1.2 Handling Warning Exceptions

The handling process for an exception classified as warning is quite similar to that for the error exception described above. There are only two differences: (1) a signal is sent to the operating system only when a *warning-handler* is specified and the *warning-handler* returns *throw*; (2) the signal sent by the function *warning* is by default *SIGUSR1* that will stop the process without

generating a core file. The *warning-handler* and warning signal of an *ErrorHandler* object may also be reset by calling functions of the *ErrorHandler* class.

## 10.2 Parameterized Array Classes

A parameterized class or parameterized type usually implements a generic data structure which may be bound with any particular data type to generate a specific class for the particular data type. For example, integer arrays can belong to a particular class of arrays which store collections of integers, and *String* arrays (*String* is a class for text strings) can belong to another particular class of arrays which store collections of *String* objects. Even though the type of entries in the two array classes are not the same, the two classes share many common operations such as array element selection since both classes are for arrays. Thus, a parameterized array class may be defined to represent generic arrays and to generate specific array classes for particular element types.

The array is a common data structure built into probably almost every general-purpose programming language. An array of a particular data type holds a collection of variables (or objects) of that type. These objects, or elements of the array, are stored in memory contiguously such that they can be accessed by indices. The weaknesses of the built-in array data structure are that: (1) usually no dynamic index checking supported by the language; (2) the size of an array is fixed once the array is allocated statically.

In many cases of engineering software development, it is not possible to estimate of the maximum required size of an array at the time that the array is created. In these cases, another data structure, the linked list, is often used to store collections of objects, the number of which may grow. However, this approach leads to other problems in that: (1) the direct access of elements by indices is expensive to use; (2) the code using linked lists has higher complexity

than and is not as expressive as that using arrays; and (3) the linked list structure is better used for ordered collections of objects, while in many cases, collections of objects may not necessarily be ordered.

The parameterized array classes developed in the present work provide the following features which avoid the weakness usually associated with the built-in array data structure:

1. Safe and flexible element selection: index of element in element selection operations may be checked.
2. Adjustable array size: the size of arrays may be adjusted either implicitly by element selection or element addition operations, or explicitly by the size reset operation.
3. Cursor: a cursor is defined for an array that may be used to locate a specific element in the array and to construct loops over elements in the array.
4. Built-in iterators: loops over elements in an array in certain orders may be performed easily by using these iterators.

Because of the lack of support for parameterized classes in the current version of C++ (version 2.0), a parametrized class is declared and implemented by using the macros of the C preprocessor. Two parameterized array classes, *ExtArray(T)* and *Bag(T)* are developed, where *T* stands for a particular data type or object class from which a specific array class is defined. For example, a *String* array class is denoted as *ExtArray(String)* which is expanded by the C preprocessor to *StringExtArray* as the name of the class. The class *Bag(T)* is derived from the *ExtArray(T)* class. These two classes are described in the following sub-sections.

### 10.2.1 The *ExtArray(T)* Class

Figure 10.3 shows a simplified version of the *ExtArray(T)* class declaration. As mentioned previously, the current version of C++ does not support parameterized classes, and the C preprocessor has to be used in the declaration and implementation. However, to avoid unnecessary complexity of the description, the declaration of the parameterized *ExtArray(T)* class is shown in the figure as an ordinary class. This also applies to the descriptions of other parameterized classes in this chapter.

#### 10.2.1.1 Class Definition

An object of a specific *ExtArray(T)* class holds the following properties as shown in Fig. 10.3

1. A pointer, *a*, to type *T* which points to the array elements in memory;
2. The size of the array, *sz*, and the size increment, *dsz*, for expanding the size of the array;
3. The actual number of elements in the array, *ec*, which may or may not be the same as the size of the array. It is also the maximum index has been used for the array;
4. A cursor, *cs*, which is the index of the current element.
5. A static member, *eh*, which is an object of the *ErrorHandler* class and shared by all objects of the specific *ExtArray(T)* class for exception handling.

Memory for elements of an initial array size is allocated and pointed to by *a* when an *ExtArray(T)* object is created. An object may be either created statically or by using the C++ operator *new*. When expanding the array size of

An excerpt from the `ExtArray(T)` class declaration

```
class ExtArray(T) {
protected:
    T* a;
    int sz, dsz;
    int ec, cs;
    static ErrorHandler eh;
public:
    ExtArray(T)(int s, int ds = 0);
    ~ExtArray(T)();
    T elem(int i);
    T& operator()(int i);
    T& operator[](int i);
    void set_size(int);
    void set_cursor(int cursor = 0);
    virtual int add(T& e);
    virtual int current(T& e);
    virtual int next(T& e);
    virtual int previous(T& e);
    virtual int loop(T& e);
    virtual int backloop(T& e);
};
```

Figure 10.3 A simplified version of the `ExtArray(T)` class declaration

the object, a new block of memory is allocated and the contents of the existing elements are copied to this new block of memory. The old memory block is released, and *a* is made to point to this new memory block. Thus, the array size of an *ExtArray(T)* object is expanded.

The *ExtArray(T)* class implements methods for array information retrieval, element selection, array size adjustment, iteration, direct array object assignment, and exception handling. Prototypes of some of these methods are shown in Fig. 10.3.

A specific *ExtArray* class for a particular element data type may be declared and implemented by using two macros respectively after the inclusion of the header file *extarrayCC.h*. Figure 10.4 shows the declaration and implementation of the *StringExtArray* class.

```
#include "stringCC.h"
#include "extarrayCC.h"

ExtArrayClassDeclaration(String)
ExtArrayClassImplementation(String)
```

Figure 10.4 Declaration and implementation of the *StringExtArray* class

#### 10.2.1.2 Element Selection and Array Size Adjustment

Elements in an array may be accessed by an index starting from 0. An element in an array may be selected or accessed by using one of the following three member or operator functions: *elem(int)*, *operator[] (int)*, and *operator() (int)*. These functions all accept the index of an element as an argument.



The member function *elem* provides a "read-only" efficient element access, and no element index checking is performed. The *operator()* provides both read and write access to an array element, and provides index checking such that the index must be in the range between 0 and *ec*. The *operator[]* also provides read and write access to an array element. Moreover, if the index passed to it is greater than the current number of elements, the current number of elements is reset to the index plus 1. If the index is greater than the current size of the array, the array will be expanded to a size larger than the index. However, if the size of an array is expanded to a size larger than a certain multiple of the current size, a warning message is issued to avoid expanding an array accidentally.

In addition to the *operator[]* discussed above, there are two other member functions used to adjust the size of an array: *add(T&)* and *set\_size(int)*. The function *add* accepts an argument of type *T* and assigns it to the location *ec*, the current number of elements, and increases *ec* by 1. The array will be extended using the array size increment if *ec* equals *sz*, the size of the array. The function *set\_size(int)* accepts an argument which is the new size of the array, and reset the array to the new size explicitly. Elements currently stored in the array will remain if the new size is larger than the current size *sz*, or else only the first new *sz* elements remain.

### 10.2.1.3 Cursor and Iterations

The cursor *cs* for an array object may be used to locate a specific element in the array and to construct loops over elements in the array. Methods are provided to set the current element to which the cursor refers to, and to move the cursor forward or backward. Loops over elements of the array forward or backward may be constructed using these methods. Two functions, *loop(T&)* and *backloop(T&)* are also provided as built-in iterators for performing loops. Iterators provide mechanisms to perform loops without forcing the clients of the

object class to depend on the implementation details of the class. Figure 10.5 shows several different ways to construct loops over array elements. The *StringExtArray* class is used for the illustration.

### 10.2.2 The *Bag(T)* Class

The parameterized array class *Bag(T)* is derived from the *ExtArray(T)* class which is defined as a *public* base of the *Bag(T)* class. Thus, any operation performed on an *ExtArray(T)* object may be used for a *Bag(T)* object. A simplified version of the *Bag(T)* class is shown in Figure 10.6.

#### 10.2.2.1 Class Definition

The major difference distinguishing the *Bag(T)* class from its base class *ExtArray(T)* is the notation of *null* elements. An element of a *Bag(T)* object may be null, while an element of an *ExtArray(T)* object may not be distinguished from being null or not.

The *ExtArray* class is more general and it does not place any requirement on a data type *T* from which a specific *ExtArray(T)* class may be defined. Ideally, an *ExtArray(T)* class may be generated from any data type *T*. The value of an element in an *ExtArray(T)* can be changed, but the element can not be deleted. Thus, an *ExtArray(T)* class should be used for the cases where an element deletion operation is not required. When such an operation is required, the *Bag* class should be used.

The *Bag* class requires a data type *T* support the notation of null object in order to define a specific *Bag(T)* class. Basic built-in data types such as *int* and *float* do not have the notation of null object and therefore can not be used to define a specific *Bag* class. Because an element of a *Bag(T)* class may be

```
StringExtArray a(n);
String t;
int i;
.... // some operations on a
```

#### **Forward loop using the method next**

```
a.set_cursor(i); // forward loops may start from the i'th element
a.current(t); // get the current element
do {
    .... // the loop body, performing operations on t
} while (a.next(t));
```

#### **Backward loop using the method previous**

```
a.set_cursor(i); // backward loops may start from the i'th element
a.current(t); // get the current element
do {
    .... // the loop body, performing operations on t
} while (a.previous(t));
```

#### **Forward loop using the iterator loop**

```
a.set_cursor(i); // forward loops may start from the i'th elementB
while (a.loop(t)) {
    .... // the loop body, performing operations on t
}
```

#### **Backward loop using the iterator backloop**

```
a.set_cursor_to_end(); // backward loops may start from the last element
while (a.backloop(t)) {
    .... // the loop body, performing operations on t
}
```

**Figure 10.5** Constructing loops over array elements

An excerpt from the `Bag(T)` class declaration

```
class Bag(T) : public ExtArray(T) {
public:
    Bag(T)(int s, int ds = 0) : (s, ds) { }; // inherits its base and does nothing
    int add(T& e);
    int remove(T& e);
    int loop(T& e);
    int backloop(T& e);
    int lookup(T& e);
    int purge(T& e);
    int shrink();
};
```

Figure 10.6 A simplified version of the `Bag(T)` class declaration

checked for whether it is null or not, an element in the object may be deleted, erased, and compared against null.

A specific `Bag` class for a particular element data type may be declared and implemented by using four macros respectively after the inclusion of the header file `bagCC.h`. Figure 10.7 shows the declaration and implementation of the `StringBag` class.

```
#include "stringCC.h"
#include "bagCC.h"

ExtArrayClassDeclaration(String)
BagClassDeclaration(String)
ExtArrayClassImplementation(String)
BagClassImplementation(String)
```

Figure 10.7 Declaration and implementation of the `StringBag` class

### 10.2.2.2 Additional Operations Defined in the *Bag(T)* Class

The *Bag(T)* class provides additional operations for array objects as shown in Figure 10.6. Using these methods, an element of a *Bag(T)* object may be removed from the array, null elements in the array of a *Bag(T)* object may be purged such that non-null elements will be stored continuously in the front portion of the array, and the size of the array may be shrunk such that only non-null elements will be stored in the array.

The member function *add(T&)* of the *ExtArray(T)* class is redefined in the *Bag(T)* class. If there is a null element in the first *ec* positions, this function will put the object of type *T* at that position, or else it calls *ExtArray(T)::add(T&)* to add the object to the *Bag(T)* object. The built-in iterators *loop(T&)* and *backloop(T&)* are also redefined such that only non-null elements will be passed back.

### 10.2.2.3 The *BagIterator(T)* class

An iterator class *BagIterator(T)* is defined for use with the *Bag(T)* class. The *Bag(T)* class has its own iterators, *loop* and *backloop* which use the internal cursor of an *Bag(T)* object to construct loops over array elements. Because these built-in iterators rely on a single internal cursor, they can not be used to construct nested loops over the elements of a *Bag(T)* object. This is the reason for the development of the *BagIterator(T)* class. The iterator class makes it possible to construct deeply nested loops over elements of a *Bag(T)* without knowing the internal storage structure of a *Bag(T)* object. Figure 10.8 shows a simplified version of the *BagIterator(T)* class declaration, and Figure 10.9 shows example uses of *BagIterator* objects.

Three iterators, *next*, *loop1*, and *loop2* are defined as iterators of a *StringBag* object, and three different loops are shown in Fig. 10.9.

An excerpt from the *BagIterator(T)* class declaration

```
class Bag(T) {
private:
    Bag(T)* bag;
    int cs, lt;
public:
    BagIterator(T)(Bag(T)& b);
    void reset(int init = 0);
    void reset(int init, int last);
    int cursor();
    int operator()(T& e);
};
```

Figure 10.8 A simplified version of the *BagIterator(T)* class declaration

### 10.3 A Parameterized Vector Class

A parameterized vector class, *Vector(T)*, is developed to represent the vector quantities often used in scientific and engineering computation. A specific vector class with a particular type of vector elements can be declared and implemented using this parameterized class. Several commonly used vector operations are supported, and a set of operators is overloaded for easy and expressive coding. Also, the index of an element in vector element selection operation may be checked to make vector operations safe. Figure 10.10 shows a simplified version of the *Vector(T)* class declaration. A specific vector class with a particular type of *T* can be declared and implemented using the parameterized class in the same way as defining a specific array class using *ExtArray(T)*.

To avoid copying every element of a vector when passing or returning a *Vector(T)* variable to or from a function and when assigning a *Vector(T)* variable to another, a separate data structure, *vector\_rep(T)*, is defined as shown in Fig. 10.10. This structure holds the elements and number of elements of a *Vector(T)* object. A *Vector(T)* object actually holds one pointer to an instance of such a structure. An instance of such a structure may be referred to by one

```

StringBag a(n);
String s, t;
StringBagIterator next(a), loop1(a), loop2(a);
....          // some operations on a

```

#### Forward loop using the iterator next

```

next.reset();          // reset the cursor of next to 0
while (next(s)) {     // get an element
  ....                // the loop body, performing operations on s
}

```

#### Nested loops

```

loop1.reset();        // reset the cursor of loop1 to 0
while (loop1(s)) {   // get the current element
  ....                // performing operations on s
  loop2.reset();     // reset the cursor of loop2 to 0
  while (loop2(t)) {
    ....              // performing operations on s and t
  }
}

loop1.reset();        // reset the cursor of loop1 to zero
while (loop1(s)) {   // loop from the first element of a
  ....
  loop2.reset(loop1.cursor()-1); // reset the cursor of loop2
  while (loop2(t)) { // loop from the current cursor of loop1
    ....
  }
}

```

Figure 10.9 Constructing loops using *BagIterator* objects

An excerpt from the `Vector(T)` class declaration

```

class Vector(T) {
protected:
    struct vector_rep(T) {
        T* v;
        int n, ref;
    } *p;
    static ErrorHandler eh;
public:
    Vector(T)(int sz, T iv = (T) 0);
    ~Vector(T)();
    T val(int i);
    T& operator()(int i);
    Vector(T)& assign(const Vector(T)& r);
    Vector(T) copy();
    Vector(T) plus(const Vector(T)& r, Vector(T)& rs);
    Vector(T) minus(const Vector(T)& r, Vector(T)& rs);
    Vector(T) negate(Vector(T)& rs);
    Vector(T) multiply(const T l, Vector(T)& rs);
    T dot(const Vector(T)& r);
    T min();
    T max();
    T average();
    T norm();
    Vector(T)& operator==(const Vector(T)& r);
    Vector(T) operator+(const Vector(T)& r);
    Vector(T) operator-(const Vector(T)& r);
    Vector(T) operator-();
    Vector(T) operator*(T r);
    T operator*(const Vector(T)& r);
    friend ostream& operator<<(ostream& os, Vector(T)& r);
    friend istream& operator>>(istream& is, Vector(T)& r);
};

```

Figure 10.10 A simplified version of the `Vector(T)` class declaration



or more *Vector(T)* variables. Thus, this structure also holds a counter (the member variable *ref*), to trace how many *Vector(T)* variables refer to an instance at a certain time.

When a *Vector(T)* object "a" is assigned by another *Vector(T)* object "b", the object "a" refers to the same instance of the data structure as "b", and the reference counter of the instance is increased by 1. When a *Vector(T)* object is deleted or is assigned by another object, the reference counter of the data structure instance is decreased by 1. When the reference counter of an instance of this data structure is decreased to zero, the instance is then deleted from memory.

Prototypes of some selected operation functions and overloaded operators of *Vector(T)* are also shown in Fig. 10.10. These functions and operators define many operations between vectors, between vectors and scalars, and for vector input/output. Two specific vector classes, *IntVector* and *RealVector* are defined in the object library for integer and floating-point number vectors respectively. Operations between matrices and *RealVector* objects are defined in the matrix classes. This will be discussed in the next chapter.

## CHAPTER 11 OBJECT CLASSES FOR FULL MATRICES

### 11.1 Introduction

Libraries for matrix manipulation are not new. Many of these types of libraries have been developed and utilized with varying degree of success. Most of the existing libraries have been developed based on conventional functional software design approaches as discussed previously in Chapter 3, and they have been implemented in procedural languages such as FORTRAN and C. These "functionally designed" libraries usually work well for scientific and engineering applications where only a limited number of matrix types and matrix operations are involved. However, procedural languages such as FORTRAN and C can not facilitate optimally the development of a general-purpose matrix library, i.e., one in which many different types of matrices and matrix operations are supported.

Ideally, a general-purpose matrix manipulation library should satisfy the following requirements:

- i. **Abstraction:** Matrices should be represented as specific "library defined" matrix types, rather than by general data types such as multi-dimensional arrays. This leads to a better conceptual clarity of matrix manipulations.
- ii. **Utilization of Matrix Characteristics:** Different characteristics such as symmetry and diagonalness should be represented by specific matrix types to save storage and to improve efficiency.
- iii. **Dynamic Creation and Destruction of Matrices:** It should be possible to create a matrix of any size and type at run-time whenever the matrix is

needed. It should also be possible to destroy a matrix and release its memory whenever there is no further need for the matrix.

- iv. **Ease of Use and Expressiveness:** Matrix manipulations in an application program that uses the library should be as close in form to the mathematical matrix expressions as possible. For example, it would be desirable to express a matrix operation as  $D = A * B + C$  in the application program, rather than use one or a series of function calls. Where function calls are needed, the names of library functions should be mnemonic.
- v. **Information Hiding:** Any information which relates to implementation details such as the manner in which the elements of a matrix are stored, should be hidden from software components using the library.
- vi. **Provision of a Standard Interface:** The library should provide a standard or unified interface for different matrix types. That is, a certain matrix operation can be coded uniquely regardless the types of the matrices operated on by the operation. This also leads to greater ease of use of the library.
- vii. **Efficiency:** Using this type of library, it should not result in significant additional run-time overhead as compared to conventional libraries.
- viii. **Extendibility:** The library should be designed in a way such that new matrix types and new operations on matrices can be added to the library without affecting existing functions.

Object-oriented programming is the most promising software technique at present for satisfaction of all the above requirements. A number of matrix manipulation packages have been implemented using object-oriented approaches and described in the recent literature (Eckel, 1989; Baugh et al., 1989; Lee, 1989). Eckel (1989) discusses an object-oriented matrix library implemented in the C++ language. However, only one matrix class is developed and this class

supports only general full matrix operations. Baugh et al. (1989) have developed an object-oriented matrix library containing several object classes for matrix operation. However, no details are given about the design and implementation techniques, the run-time efficiency, or their implementation language. Lee (1989) developed a matrix class in C++ in his work. This will be discussed further in the next section.

A set of matrix classes for full and sparse matrices are developed in the present work. This chapter describes the classes for full matrices, and the next chapter describes the classes for sparse matrices. Section 11.2 describes pitfalls encountered in the implementation of matrix libraries using conventional procedural languages. Section 11.3 provides an overview of the classes for full matrices. Section 11.4 presents the design of the *Matrix* class which is the base class for other full matrix classes. Two example derived classes of the *Matrix* class are described in Section 11.5.

## 11.2 Procedural Libraries for Full Matrices

This section discusses techniques and pitfalls involved with the implementation of a matrix manipulation library using the procedural languages FORTRAN and C.

### 11.2.1 Abstraction: the Representation of a Matrix

The FORTRAN language lacks structure data types. A matrix in FORTRAN is usually represented as a one or two dimensional array. The dimensions of the matrix are represented separately by integer variables or constants. These integer variables need to be passed to library routines together with the array to perform a certain operation. For example, the specification for a FORTRAN subroutine that performs the matrix multiplication  $C = A * B$  can

be written as

```
SUBROUTINE MMULTM(A, B, C, M, K, N)
  INTEGER M, K, N
  DIMENSION A(M, K), B(K, N), C(M, N)
```

To use this subroutine, a program must pass the correct dimension parameters (i.e., the integer variables *M*, *K*, *N*) together with the arrays to the subroutine, and also it must pass in a correct order.

However, in the C language, a data structure can be defined to represent a matrix:

```
typedef struct _MatrixRec {
  int row;
  int col;
  double **a;
} MatrixRec, *Matrix;
```

Thus, a variable of type *MatrixRec* or *Matrix* is an abstract representation of a matrix. This abstraction contains the dimensions of the matrix, and once a matrix variable is created, the programmer does not have to pass the dimensions of the matrix explicitly to the library functions. The specification of a C library function performing the matrix multiplication operation can be written as

```
int matrix_mult_matrix(a, b, c)
  Matrix a, b, c;
```

The dimensions of the matrices can be checked in the function to determine if the multiplication can be performed. The function will return 1 if the operation is successful and 0 for failure.

Alternatively, this function may be defined as a function returning a *Matrix*

```

Matrix matrix_mult_matrix(a, b)
Matrix a, b;

```

The matrix holding the result of the multiplication would be dynamically created in the function and then returned from the function. However, allocating memory in a library function without an appropriate method ensuring that the memory is eventually freed would be disastrous if the function is called in a repeated manner.

### 11.2.2 Dynamic Creation and Destruction of a Matrix

The FORTRAN 77 standard does not support dynamic memory allocation. Thus, the coding of matrix operations with matrices whose dimensions can only be determined at run time is complicated. Programmers are forced to handle dynamic memory allocation in the application. This is usually done by dividing a large static array into many small blocks during the program execution. Programmers must allocate memory not only for the matrices appearing in the matrix expression to be evaluated, but also memory must be allocated for the matrices holding possible intermediate results. For example, the FORTRAN code evaluating the matrix expression  $E = A * B + C * D$  can be written as

```

COMMON TEMPSTORAGE(10000)
....
CALL MEMORYALLOC(M*N, NTMP1)
CALL MEMORYALLOC(M*N, NTMP2)
CALL MMULTM(A, B, TEMPSTORAGE(NTMP1), M, K, N)
CALL MMULTM(C, D, TEMPSTORAGE(NTMP2), M, K, N)
CALL MADDM(TEMPSTORAGE(NTMP1),
           TEMPSTORAGE(NTMP2), E, M, N)
CALL MEMORYFREE(M*N, NTMP1)
CALL MEMORYFREE(M*N, NTMP2)

```

In the above code, *MEMORYALLOC* and *MEMORYFREE* are utility subroutines implementing the dynamic memory allocation in the large array

*TEMPSTORAGE*. As it can be seen from the above code, memory for matrices A, B, C, D, and E, as well as for the intermediate results of  $A * B$  and  $C * D$  needs to be allocated. This has to be done explicitly in the code. Too much coding effort is involved here. Of course a library routine can be specifically designed for evaluating such a matrix expression. Such kinds of routines can be found in many FORTRAN libraries. However, many functions of this sort, which perform certain "composite" operations, would be needed in a general-purpose library. This type of library is not ideal because application programmers must find their way through a maze of functions. The goal of the present work on matrix classes is to develop a general-purpose library avoiding the use of a large number of special purpose functions.

The C language supports dynamic memory allocation. Thus, creation and destruction of a matrix variable is easy to perform in C. The code that evaluates the expression  $E = A * B + C * D$  can be written as

```
tmp_matrix1 = new_matrix(a->row, b->col);
tmp_matrix2 = new_matrix(c->row, d->col);
matrix_mult_matrix(a, b, tmp_matrix1);
matrix_mult_matrix(c, d, tmp_matrix2);
matrix_add_matrix(tmp_matrix1, tmp_matrix2, e);
free_matrix(tmp_matrix1);
free_matrix(tmp_matrix2);
```

However, the programmer has to create and destroy the temporary matrices explicitly in the code.

### 11.2.3 Utilization of Matrix Characteristics

Compared with other programming languages, FORTRAN does not optimally support the representation of matrix characteristics such as symmetry and diagonalness to save storage and improve efficiency. In many FORTRAN programs, for example, a symmetric matrix is represented either by a two-

dimensional or a one-dimensional array. If coded in the first way, the symmetry of the matrix is not utilized to save storage. If coded in the second way, the matrix element selection from the one-dimensional array is complicated. Library routines can be developed based on the one-dimensional array representation. However, in general, application programmers have to understand the representation in order to allocate the required memory and to access elements of the matrix.

If the C language is used, data structures can be defined to represent matrices with different characteristics. For example, a data structure *SymMatrix* can be defined to represent symmetric matrices where only the diagonal elements and elements in the lower triangle are stored. A data structure *DiagMatrix* can be defined to represent diagonal matrices where only the diagonal elements are stored. Implementation details of the manipulation of such data structures can be hidden in a matrix library. However, we then face the problem of how to develop library functions to handle mixed operations on matrices of different types.

General library functions can be developed for specific operations involving matrices of different types. For example, a function *matrix\_mult\_matrix(a, b, c)* can be developed for matrix multiplication, where *a* and *b* can be matrices of any type, and the type of matrix *c*, which holds the result, depends on the types of *a* and *b*. This function can be implemented using a set of *switch* and *case* instructions to figure out how the operation be performed. However, this leads to a complex and inefficient implementation. Moreover, the addition of a new matrix type to the library will mean the modification and recompilation of all such functions and may introduce new bugs into the library.

An alternative approach is to develop a set of functions, and have each of these functions handle a special case. For example, a function *dmatrix\_mult\_smatrix(a, b, c)* might handle a diagonal matrix multiplied by a



symmetric matrix, and *smatrix\_mult\_dmatrix(a, b, c)* might handle a symmetric matrix multiplied by a diagonal matrix. A very large number of functions would need to be developed to handle all possible combinations by this approach. This kind of library would not be very usable because programmers have to find their way through a maze of functions.

#### 11.2.4 Ease of Use and Expressiveness

According to the above discussion, none of the libraries implemented in FORTRAN or C can be said to provide ease of use in general. Moreover, the expressiveness of the code cannot be achieved unless the implementation language allows operators such as "+", "-", and "\*" to be redefined for matrix types. This, of course, cannot be done in C or FORTRAN.

In summary, matrix libraries implemented in the FORTRAN or C language may work well for scientific and engineering applications where only a limited number of matrix types and matrix operations are involved. However, these two languages do not provide enough support for the development of a general-purpose matrix manipulation library where many different matrix types and many different operations are supported. At the present time, use of object-oriented software design methodology and object-oriented languages seems to be the most promising approach for the implementation of such a library.

However, designing a matrix library in terms of objects and implementing the library in an object-oriented language do not necessarily result in a better library. Lee (1989) designed a matrix class in his work. A simplified version of the matrix class declaration from (Lee, 1989) is shown in Figure 11.1.

**An excerpt from the MATRIX class declaration**

```
enum MAT_TYPE { MAT_FULL, MAT_SYMMETRIC,  
                MAT_LOWER, MAT_UPPER, MAT_BAND };  
  
class MATRIX {  
private:  
    OB_NAME      obName;  
    MAT_TYPE     matType;  
    INTEGER      noRow, noCol;  
    INTEGER      bandwidth;  
    INTEGER      size;  
    REAL*        body;  
public:  
                MATRIX(char*, MAT_TYPE, INTEGER, INTEGER, char*);  
                ~MATRIX();  
    PROCEDURE    Init(REAL value, ...);  
    PROCEDURE    Save(BLOCK& block, INTEGER& cursor);  
    PROCEDURE    Load(BLOCK& block, INTEGER& cursor);  
    MATRIX&      operator = (REAL R);  
    MATRIX&      T();  
};
```

Figure 11.1 A simplified version of the *MATRIX* class declaration  
(from Lee, 1989)

As shown in the figure, a single class *MATRIX* is defined for matrices with different characteristics such as symmetric. Thus, *switch* and *case* instructions have to be used in the implementation. Problems with this approach have been discussed previously. An effort has been made in this work to study the problem and to design and implement such a library following the object-oriented methodologies.

### 11.3 Overview of Full Matrix Classes

#### 11.3.1 Classification

Six object classes for representing full matrices are developed in the present work. These classes are:

1. The *Matrix* class, which represents full matrices that do not have any special characteristics. This class also serves as the base class for other specific matrix classes. It maintains the properties that are common to all matrix classes, such as the dimensions of a matrix. It implements methods that are common to all matrix classes.
2. The *DMatrix* class, which represents diagonal matrices. Only the diagonal elements are stored for this type of matrix.
3. The *SMatrix* class, which represents symmetric matrices. Only the diagonal elements and the lower triangle elements are stored in this case.
4. The *LTMatrix* class, which represents lower-triangle matrices. Only the diagonal and lower triangle elements are stored.
5. The *UTMatrix* class, which represents upper-triangle matrices. Only the diagonal and upper triangle elements are stored.
6. The *LUMatrix* class, which is used to create objects for storing the decomposed form LU for any type of matrix. The matrix factor L is

stored in the diagonal and lower triangle of a *LUMatrix* object, and the matrix factor *U* is stored in the upper triangle.

The type of matrix elements is defined as *Real* and can be specified as either *float* or *double* at compilation time.

### 11.3.2 Matrix Manipulation Functions

Functions in the matrix classes for manipulation of full matrices may be classified either in terms of member functions and friend functions or in terms of operation functions and overloaded operator functions. These functions can be further classified according to the operations they perform:

1. **Constructors, Destructors and Coercion Functions:** Constructors of a matrix class are used for creation and initialization of matrix objects of that class. Only one destructor can be defined for a class. The destructor is used to clean up and destroy matrix objects of that class. For each matrix class, a coercion function is defined and used to convert objects of other matrix classes to objects of that class.
2. **Element Selection Functions:** These functions provide ways to access an element, a row of elements, or a column of elements in a matrix directly.
3. **Matrix Operation Functions:** These functions perform operations on matrices such as addition, subtraction, multiplication, decomposition, and taking the inverse, as well as matrix and vector mixed operations. These functions may be of the type of a certain matrix class, of the *RealVector* type, or of the *Real* type depending on the operation. A function is said to be of a certain type when the function returns an object of that type.
4. **Utility Functions:** Utility functions are used for matrix I/O, error handling, and other miscellaneous operations.

In the design of these functions, if more than one function performs a similar operation on objects of different matrix classes, these functions are given syntactically the same specification. For example, the operation of computing the product of a scalar with a matrix is defined in most matrix classes, and the specification of all these functions are defined in the following form:

```
matrix::product(matrix& lp, const Real rp);
```

where *matrix* stands for the name of a specific class to which the *product* operation function is defined.

### 11.3.3 Use of Two Interfaces for Matrix Operations

The results of many matrix operation functions are matrix objects of certain classes. Two rules are followed in the design of such matrix operation functions: (1) a function which returns an object of a certain matrix class should be defined as a member function of that class; and (2) the matrix object through which a member function of this type is invoked holds the result of the operation. For example, the result of multiplying a scalar with a symmetric matrix is a symmetric matrix. The function that implements this operation should be defined as a member function of the *SMatrix* class. Thus, the prototype of this function is

```
SMatrix SMatrix::product(SMatrix& lp, Real rp);
```

This function can be used in the form

```
s.product(sm, c);
```

where *sm*, a *SMatrix* object, and *c*, a scalar, are the operands, and *s*, a *SMatrix* object, holds the result of the operation. These rules facilitate the implementation of operator overloading for matrix operations.

Many operators such as +, -, \*, /, += are overloaded for matrix classes. As a result, matrix operations can be coded in a more abstract and expressive

fashion. Thus, there are two types of functions defined in a matrix class: operation functions and operator functions. An operation function of a class is invoked through an object of the class by referencing the name of the function. The function *SMatrix::product* described above is a typical example of an operation function. An operator function is invoked by applying an operator to an object of the class. In the present development, operator functions are implemented based on operation functions. Thus, there are two interfaces for most matrix operations based on one implementation. One interface consists of operation functions and is referred to as the function interface. The other uses operator functions and is referred to as operator interface.

For example, the operator `*` is overloaded to perform the operation of multiplying a scalar with a symmetric matrix as discussed above. This is done by defining two operator functions: one is a member function and the other is a friend function of the *SMatrix* class. The prototype of the two operator functions is

```
SMatrix SMatrix::operator*(const Real rp);
friend SMatrix operator*(const Real lp, SMatrix& rp);
```

Two operator functions are defined because C++ invokes a member operator function based on the left operand. Thus, if *s* is a *SMatrix* object and *c* is a *Real* variable, the operation *s \* c* invokes the member operator function. For the operation *c \* s*, a friend operator function should be defined. The two operator functions are implemented by simply creating a temporary *SMatrix* object and then invoking the operation function *product*

```
SMatrix SMatrix::operator*(const Real rp)
{
    SMatrix rs(p->row);
    return rs.SMatrix::product(*this, rp);
}
```

```

SMatrix operator*(const Real lp, SMatrix& rp)
{
    SMatrix rs(rp.p->row);
    return rs.SMatrix::product(rp, lp);
}

```

In the member operator function, *\*this* refers to the object by which the function is invoked.

The differences in the use of the two interfaces are as follows:

1. The operator interface is much easier to use than the function interface because programmers do not have to remember function names to code matrix operations. Also, by using the operator interface, the code will be more expressive and abstract.
2. Programmers are forced to create temporary objects to hold operation results if operation functions are used. Temporary objects are automatically created when operator functions are used. These temporary objects will be automatically destroyed when they are no longer needed.
3. Because a temporary object is created each time that an operator function is called (this is the case for most operator functions), more CPU-time is needed for using an operator function than for using the corresponding operation function.

If an application is computationally intensive and the efficiency largely depends on certain matrix operations, the function interface should be used for these operations. Otherwise the operator interface is recommended.

#### 11.3.4 Extensibility

These matrix classes are readily extensible. New classes for matrices with other characteristics such as tri-diagonalness may be developed as derived classes

of the *Matrix* class. New operations such as eigenvalue and eigenvector computation may be implemented to the existing classes. No modification of the existing classes is necessary for such extensions. Also, the development of a new matrix class is eased by taking the *Matrix* as a base class because only operations that take the advantage of the characteristics of matrices the new class represents need to be implemented.

#### 11.4 Design of the *Matrix* Class

This section describes the design details of the *Matrix* class. The *Matrix* class represents general full matrices and implements operations on general matrices. Figure 11.2 shows a simplified version of the *Matrix* class declaration, where the prototypes of some member functions are not shown for conciseness.

##### 11.4.1 Matrix Representation

The *Matrix* class declaration contains the representation for a general matrix object, i.e., a pointer to a data structure *matrix\_representation* as shown in Fig. 11.2. This structure contains the following information about a matrix: (1) *row* and *col* are number of rows and columns; (2) *ref* is the reference count to be discussed below; and (3) *a* is a pointer to an array of pointers which is dynamically allocated. Each entry in the array is a pointer to matrix elements in a row which are also dynamically allocated. *Matrix* objects only contain a pointer to this structure. This representation is adopted from the matrix class developed by Eckel (1989).

The representation is inherited by the classes derived from *Matrix*. The only difference between different matrix classes is that only the necessary storage for the matrix elements is allocated and pointed to by *a*. The amount of memory allocated and the way the elements being stored depends on the



An excerpt from the *Matrix* class declaration

```

class Matrix {
protected:
    struct matrix_representation {
        int row, col, ref;
        Real** a;
    } *p;
    static ErrorHandler eh;
public:
    Matrix(int rc, int cc, Real* iv); // constructor

    Matrix(const Matrix& x); // copy-constructor
    ~Matrix(); // destructor
    virtual Real val(int i, int j);
    virtual Real& operator()(int i, int j);
    Real* operator[](int i);
    Matrix assign(Matrix& rp);
    Matrix plus_assign(Matrix& rp);
    Matrix multiply_assign(const Real rp);
    Matrix negate(Matrix& rp);
    Matrix transpose(Matrix& rp);
    Matrix sum(Matrix& lp, Matrix& rp);
    Matrix difference(Matrix& lp, Matrix& rp);
    Matrix product(Matrix& lp, Matrix& rp);
    RealVector multiply_rv(RealVector& rp, RealVector& rs);
    Matrix quotient(Matrix& lp, const Real rp);
    Matrix operator==(const Real rp);
    Matrix operator==(Matrix& rp);
    Matrix operator+=(Matrix& rp);
    Matrix operator*=(const Real rp);
    Matrix operator-();
    Matrix operator/(Real rp);
    Matrix operator*(Real rp);
    Matrix operator~();
    friend Matrix operator*(const Real lp, Matrix& rp);
    friend Matrix operator^(RealVector& lp, RealVector& rp);
    friend Matrix operator+(Matrix& lp, Matrix& rp);
    friend Matrix operator*(Matrix& lp, Matrix& rp);
    friend RealVector operator*(Matrix& lp, RealVector& rp);
    virtual LUMatrix decompose(LUMatrix& rp, Real tol = (Real) 0.0001);
};

```

Figure 11.2 A simplified version of the *Matrix* class declaration

characteristics of each matrix class. This is referred to as the storage architecture of a matrix class. For the *Matrix* class, memory is allocated for every element of a full matrix.

Many matrix operation functions return a *Matrix* object which holds result of the operation. Temporary *Matrix* objects are often generated by operator functions. *Matrix* objects may be also passed by value to a function (however, matrix objects are always passed by reference for all matrix classes developed in the present work). By adopting this representation when matrix objects are passed to and returned from functions, only a pointer is transferred to or from the functions. This leads to a simple and efficient implementation.

However, the problem with this representation is that care must be taken to release the memory in the free storage occupied by a "dead" object and not to release the memory from the free storage utilized by "live" objects. This task is accomplished by the use of the field *ref* in the structure *matrix\_representation*. The field *ref* is called the reference count. It indicates how many objects are currently using a particular instance of the data structure.

When a matrix object is created, for example as a temporary object, the memory for an instance of the structure is allocated according to the dimensions and the class of the object, and the reference count of the instance is set to 1. When this object is assigned to another object of the same matrix class, the object being assigned will refer to the instance of the structure and the reference count is increased by 1. This avoids the copying of all matrix elements for simple assignments. A member function, *copy*, is provided in each matrix class to actually make a copy of a matrix object when desired. When a matrix object is destroyed or the memory reference of the object is changed, the reference count of the associated instance of this data structure is decreased by 1. When the reference count of an instance of this structure is decreased to zero, the instance will be deleted.

### 11.4.2 Matrix Operations

Table 11.1 summarizes the operator and the function interface for the operations implemented in the *Matrix* class. Because the *Matrix* class serves as the base class for other specialized matrix classes, the operations implemented in this class are inherited by derived classes of *Matrix*. Mixed operations between matrix objects of different classes must be taken into account in the design and implementation of these operation functions.

A derived class may define its own member and friend functions which have the same names as those defined in *Matrix* class. Such functions are considered as different functions than the ones defined in *Matrix*. When an operation or operator function is invoked through an object of a derived class, the function defined in the *Matrix* class will be used if the function is not defined in the derived class. If the function is not defined in *Matrix* class either, the function call generates an error at compilation time. If a function defined in *Matrix* is used to operate on a matrix object of a derived class, no advantage can be taken from special characteristics of the derived class such as symmetry. Operations that take advantage of the characteristics of a certain derived matrix class should be defined in the derived class. For example, the addition of a symmetric matrix and a diagonal matrix results in a symmetric matrix. Such an operation, if it is desired, should be defined in the *SMatrix* class.

According to the above discussion, the operation functions implemented in the *Matrix* class may be used to operate on objects of derived classes of which the actual storage architectures are unknown to *Matrix*. This causes a problem in the definition of these functions. The problem is how these functions work on objects having unknown storage architectures. The key to solving this problem is the *virtual* function mechanism of the C++ language.

As can be seen from Figure 11.2, the operation function *val(int i, int j)* and the operator function *operator()(int i, int j)* are virtual. These two functions are

Table 11.1. Interface of the *Matrix* class

Operator Interface	Function Interface	Explanation
$M(i, j)$	<code>M.val(i, j)</code>	element selection
$A[i]$	----	matrix row selection
$M = A$	<code>M.assign(A)</code>	coercion function
$M = c$	<code>M.assign(c)</code>	assigning the same value to all elements of $M$
$M += A$	<code>M.plus_assign(A)</code>	adding matrix $A$ to $M$
$M -= A$	<code>M.minus_assign(A)</code>	substrating matrix $A$ from $M$
$M *= c$	<code>M.multiply_assign(c)</code>	multiplying a scalar $c$ to $M$
$-A$	<code>T.negative(A)</code>	negative of matrix $A$ .
$A / c$	<code>T.quotient(A, c)</code>	dividing matrix $A$ by a scalar $c$
$A * c$	<code>T.product(A, c)</code>	multiplying matrix $A$ with a scalar $c$
$A^T$	<code>T.transpose(A)</code>	matrix transposition.
$c * A$	<code>T.product(A, c)</code>	multiplying a scalar $c$ with matrix $A$
$v^T u$	<code>T.assign(v, u)</code>	$T = vu^T$
$A + B$	<code>T.sum(A, B)</code>	matrix summary.
$A - B$	<code>T.difference(A, B)</code>	matrix difference.
$A * B$	<code>T.product(A, B)</code>	matrix product.
$A * v$	<code>A.multiply_rv(v, t)</code>	right-multiplying vector $v$ with matrix $A$
$v * A$	<code>A.multiply_lv(v, t)</code>	left-multiplying vector $v$ with matrix $A$
----	<code>M.make_copy(A)</code>	copying $A$ to $M$
----	<code>M.copy()</code>	generating a copy of $M$ in a temporary <i>Matrix</i> object
----	<code>M.decompose(L, tol)</code>	decomposing matrix $M$
----	<code>M.decompose(tol)</code>	decomposing matrix $M$ and storing it in a temporary <i>LUMatrix</i> object.
----	<code>triple_product(v, A, u)</code>	triple product $v^T Au$ .

Symbols:  $M$ : an object of *Matrix* class.  $T$ : a temporary object of *Matrix* class.  
 $A, B$ : objects of any matrix class.  $L$ : an object of *LUMatrix* class.  
 $u, v, w$ : objects of *RealVector* class.  $t$ : a temporary object of *RealVector* class.  
 $c$ : a scalar variable of type *Real*.

used for selecting elements of a matrix. The two parameters  $i$  and  $j$  are the row and column indices of a matrix element respectively. Both functions are declared *inline* for efficiency. The usages of the two functions are different. The function *val* returns the value of the element  $(i,j)$  without index checking. Moreover, if the element referred to by the two indices is actually zero (e.g., if  $j > i$  in a reference to an element of a lower-triangle matrix), zero will be returned. The operator function *operator()* returns a reference to the element being selected with index checking. Thus, the call to this function can be used as an *lvalue* to change the contents of the element being referred to. Moreover, if the element referred to by the two indices is actually zero and is not stored according to the storage architecture of the object, an error will be reported by calling the error reporting function *error* through the static member *eh*.

The two functions, *val* and *operator()*, must be defined by a derived class if the derived class uses a different storage architecture from the *Matrix* class. An operation function of the *Matrix* class uses the function *val(int i,int j)* to access elements of a matrix object passed to the function, if the object may be of a derived class. Moreover, the coercion function of each matrix class also uses the function *val* to convert a matrix object of any other class to an object of its class. Thus, by the use of the virtual function mechanism, these functions are able to operate on objects with unknown storage architectures. This facilitates the implementation of mixed operations between different matrix classes.

### 11.5 Design of Derived Classes

Two derived classes of the *Matrix* class, the *SMatrix* class for symmetric matrices and the *LUMatrix* class for objects which store decomposed matrices, are described in this section.

### 11.5.1 The *SMatrix* Class

The *SMatrix* class represents symmetric matrices. It inherits the properties defined in the *Matrix* class, and it does not define any new properties of its own. Only the diagonal and lower-triangle elements of a symmetric matrix are stored in a *SMatrix* object. Because the function prototypes contained in the *SMatrix* class declaration are very similar as those in the *Matrix* class declaration, the *SMatrix* class declaration is not shown here.

In addition to the operation functions inherited from the *Matrix* class, the *SMatrix* class defines a number of operation functions utilizing the symmetry characteristic to achieve computational and storage efficiency. Table 11.2 lists some typical operation functions of the *SMatrix* function interface and the corresponding operator interface. One important component that should be highlighted is the coercion function of the *SMatrix* class, *SMatrix::assign(Matrix&)*. This function is used when a matrix object of any class is assigned to a *SMatrix* object. Only the diagonal and lower-triangle elements of a matrix object are copied to the *SMatrix* object. The elements of the upper-triangle of the *SMatrix* are determined by symmetry.

### 11.5.2 The *LUMatrix* Class

The *LUMatrix* is a special matrix class used solely for creating objects to hold the decomposed form of matrix objects of any other class. Assumptions are made in the design of the *LUMatrix* that: (1) a full matrix is needed to store the decomposed form of a matrix of any class; and (2) the solution of linear simultaneous equations and calculation of the inverse of a matrix can only be performed through a *LUMatrix* object that stores the decomposed form of the matrix. These assumptions lead to a standard interface for solving linear simultaneous equations and for matrix inversion. A simplified version of the *LUMatrix* class declaration is shown in Figure 11.3.

Table 11.2 Interface of the *SMatrix* class

Operator Interface	Function Interface	Explanation
$S(i, j)$	<code>S.val(i, j)</code>	element selection
$S = c$	<code>S.assign(c)</code>	assigning a scalar $c$ to all elements of $S$
----	<code>S.assign(v)</code>	$S = vv^T$
$S = Y$	<code>S.assign(Y)</code>	assigning an <i>SMatrix</i> to another
$S = M$	<code>S.assign(M)</code>	coercion function
$S += Y$	<code>S.plus_assign(Y)</code>	adding a <i>SMatrix</i> to another
$S -= Y$	<code>S.minus_assign(Y)</code>	substrating a <i>SMatrix</i> from another
$S * c$	<code>S.multiply_assign(c)</code>	multiplying a scalar to a <i>SMatrix</i>
----	<code>S.product_assign(M)</code>	$S = M^T M$
$S + D$	<code>T.sum(S, D)</code>	sum of a <i>SMatrix</i> and a <i>DMatrix</i> .
$S - D$	<code>T.difference(S, D)</code>	difference of a <i>SMatrix</i> with a <i>DMatrix</i>
$D - S$	<code>T.difference(D, S)</code>	difference of a <i>SMatrix</i> with a <i>DMatrix</i>
$S - Y$	<code>T.difference(S, Y)</code>	difference of two <i>SMatrix</i> objects.
$S * c$	<code>T.product(S, c)</code>	multiplying a <i>SMatrix</i> with a scalar
$c * S$	<code>T.product(S, c)</code>	multiplying a <i>SMatrix</i> with a scalar
$S / c$	<code>T.quotient(S, c)</code>	dividing a <i>SMatrix</i> by a scalar
----	<code>S.triple_product(M, S)</code>	$S = M^T S M$
----	<code>S.triple_product(M, D)</code>	$S = M^T D M$
----	<code>S.decompose(L, tol)</code>	decomposing $S$ and stored in $L$

Symbols:  $M$ : an object of any matrix class.

$S, Y, R$ : objects of *SMatrix* class.  $T$ : a temporary object of *SMatrix* class.

$D$ : an object of *DMatrix* class.  $L$ : an object of *LUMatrix* class.

$v$ : objects of *RealVector* class.

$c$ : a scalar variable of type *Real*.

**An excerpt from the LUMatrix class declaration**

```
class LUMatrix : private Matrix {
public:
    short bad;
    LUMatrix(int sz);
    LUMatrix(const LUMatrix& x);
    int row_count();
    int col_count();
    int is_bad();
    LUMatrix assign(LUMatrix& rp);
    LUMatrix operator=(LUMatrix& rp);
    Real determinant();
    Matrix inv(Matrix& rs);
    Matrix inverse();
    RealVector solve(RealVector& rp, RealVector& rs);
    friend RealVector operator/(RealVector& lp, LUMatrix& rp);
    void print(ostream& os = cout);
    int read(istream& is = cin);
};
```

**Figure 11.3** A simplified version of the *LUMatrix* class declaration



A new member variable, *bad*, is defined in the *LUMatrix* class as shown in the figure. This variable is a flag indicating whether the decomposition of a matrix, of which the *LUMatrix* object is the result, is successful or not. Unlike other derived classes for which the *Matrix* class is the *public base class*, the *Matrix* class is declared as a *private base class* of *LUMatrix*. Thus, none of the operation or operator functions defined in *Matrix* can be applied to a *LUMatrix* object. This avoids possible meaningless operations on an *LUMatrix* such as adding two decomposed matrices. Only the operation functions and operator functions defined in the *LUMatrix* class can be used to operate on an *LUMatrix* object. These operations include : (1) solving linear simultaneous equations; (2) inverting a matrix; and (3) calculating the determinant of a matrix.

It should be noted that in the definition of the classes *Matrix* and *SMatrix*, there are exceptions to the rule stated previously that a function which returns an object of a certain matrix class should be defined as a member function of that class. The functions which decompose a *Matrix* object or a *SMatrix* object and return a *LUMatrix* object are not defined as member functions of the *LUMatrix* class. Instead, they are defined as member functions of the *Matrix* and *SMatrix* classes respectively. The reason for doing this is that the matrix decomposition process cannot be performed efficiently without knowing the storage architecture of the matrix class.

To illustrate the use of the *LUMatrix* class, Figure 11.4 shows an example program for solving linear simultaneous equations. This example program also shows the abstract nature and the expressiveness of coding with these matrix classes.

## 11.6 Benchmark Testing and the Matrix Calculator

Two sets of benchmark tests have been conducted on a Sun 3/60 workstation to test the efficiency of the matrix classes developed in the present

```
#include "matrixCC.h"

int
main()
{
    int n;           // dimension of the matrix
    cin >> n;       // read the dimension
    Matrix M(n, n); // create a Matrix object
    RealVector R(n); // create a RealVector object as the right-hand-side vector
    cin >> M;        // read in the matrix from standard input
    LUMatrix L = M.decompose(); // create a LUMatrix to hold the decomposed matrix
    if (!L.is_bad()) { // if the decomposition is successful
        cin >> R;      // read in the right-hand-side vector
        RealVector S = R / L; // solve the equation and create a
                               // RealVector object to hold the solution
        cout << "The solution" << S; // print the solution
        cout << "Verify the solution" << M*S; // verify the solution
    } else {
        cout << "Singular Matrix";
    }
}
```

Figure 11.4 An illustration program of the *LUMatrix* class

work. Programs have been implemented in the FORTRAN, C and C++ languages for a number of matrix operations. In the C++ implementation, both the function and operator interfaces have been used. All calculations in the benchmark tests have been performed using double precision. All these programs are compiled without any of the optimization options of the FORTRAN, C, or C++ compilers. The C++ programs for the second test case are listed in Figure 11.5. Complete listings of other test programs can be found in (Zhang et al., 1990a).

The first test is to evaluate the matrix expression  $S = D * B$  for 2000 times, where the matrix D is 5 by 5, and B is 5 by 10. Table 11.3 compares CPU times used by the programs.

The second test is to evaluate the matrix expression  $S = B^TDB$  for 1000 times, where the matrix D is 5 by 5 and symmetric, and B is 5 by 10. The symmetry characteristics of the matrices S and D are utilized in the computation. Table 11.4 compares CPU times used by these programs.

From the testing results it can be seen that the programs implemented in C++ using the function interface are slower in both cases than the programs implemented in FORTRAN and C. In the first case, the C++ program uses 24% and 46% more time than the FORTRAN and C programs respectively, and in the second case, it uses 17% and 23% more time than the FORTRAN and C programs respectively. This is a reasonable price to pay for the benefits of using the C++ language.

Also it can be seen that the operator interface is less efficient than the function interface. The reason for this is the creation and destruction of temporary matrices. A temporary matrix is created and destroyed each time when a matrix expression is evaluated in the first case, and two temporary matrices are created and destroyed in the second case. Moreover, in the second case, the symmetry characteristic of the matrix D and the resulting matrix is not

**Benchmark testing of the function interface**

```
#include "matrixCC.h"
main()
{
    int m, n, t;
    cin >> m >> n >> t;
    Matrix b(n, m);
    SMatrix d(n, n);
    SMatrix s(m, m);
    cin >> b >> d;
    int i = t;
    while (i--)
        s.triple_product(b, d);
}
```

**Benchmark testing of the operator interface**

```
#include "matrixCC.h"
main()
{
    int m, n, t;
    cin >> m >> n >> t;
    Matrix b(n, m);
    SMatrix d(n, n);
    SMatrix s(m, m);
    cin >> b >> d;
    int i = t;
    while (i--)
        s = b * d * b;
}
```

Figure 11.5 The C++ benchmark testing programs for Case 2

Table 11.3 Efficiency of  $S = D * B$  (Case 1, 2000 operations)

Language	CPU (seconds)
FORTRAN	33
C	28
C++ (function interface)	41
C++ (operator interface)	45

Table 11.4 Efficiency of  $S = B^T D B$  (Case 2, 1000 Operations)

Language	CPU (seconds)
FORTRAN	42
C	40
C++ (function interface)	49
C++ (operator interface)	84

utilized. The evaluation of the right-hand-side of the expression results in a full matrix. Then the full matrix is assigned to the symmetric matrix  $S$  by the coercion function of the *SMatrix* class.

A matrix calculator has been developed using the matrix library. This program named *Mac* is used to perform matrix operations in an interactive mode. The initial purpose for the development of *Mac* is to test the implementation of these matrix classes. A comprehensive testing for these classes cannot be performed without a general tool. However, such a tool is also convenient and useful for doing simple matrix related calculations. Figure 11.6 is a script showing the use of *Mac* for solving a linear simultaneous equation. The text after the double hyphens "--" is an explanation of the script. A complete listing of the syntax of the *Mac* user-interface can be found in (Zhang et al., 1990a). The operator interface of the matrix classes is used in the implementation of *Mac*.

```

% mac                               -- starts Mac
mac > smatrix s0(4)                  -- defines a symmetric matrix s0R
mac > read s0                         -- input s0
4.
2. 5.
0. 2. 8.
3. 2. 1. 4.
mac > lumatrix ls(4)                 -- defines a decomposed matrix ls
mac > ls = decompose s0              -- decomposes s0 and stores it in ls
mac > vector v1(4)                   -- defines a vector v1
mac > read v1                         -- input v1
2. 2.5 3.3 4.5
mac > vector x1(4)                   -- defines a vector x1
mac > x1 = v1 / ls                   -- solve the equation and save solution in x1
mac > print x1                       -- print x1
Vector (4) ref = 2
0: -0.678571
1: 0.042857
2: 0.278571
3: 1.542857
mac > print s0 * x1                  -- print s0 * x1
Vector (4) ref = 1
0: 2.000000
1: 2.500000
2: 3.300000
3: 4.500000
mac > print x1 * s0                  -- print x1 * s0
Vector (4) ref = 1
0: 2.000000
1: 2.500000
2: 3.300000
3: 4.500000
mac > bye                           -- exit Mac
%
```

Figure 11.6 A script shows the use of the matrix calculator *Mac*

## CHAPTER 12 OBJECT CLASSES FOR SPARSE MATRICES

### 12.1 Introduction

Many scientific and engineering programs employ software components that manipulate sparse matrices. These components are often intertwined with other components. Problems associated with this approach are: (1) the matrix manipulation components are not readily reusable by other software because knowledge about the internal details of a component are necessary in order to use it; (2) the component can not be easily replaced by a different component, and, therefore, the software can not utilize new techniques for sparse matrix manipulation easily. Sparse matrix manipulation components based on different storage schemes usually lack a unified or standardized interface that would make these components interchangeable.

Therefore, two main principles should be emphasized among others in the design of software components for sparse matrix manipulation: encapsulation and provision of a standard interface. Implementation details about a certain storage scheme are hidden such that its use does not require any knowledge of its implementation. Changes in sparse matrix components will therefore not affect other components. A standard interface containing a unique correspondence between matrix manipulations with function specifications among different components implementing different schemes makes these components interchangeable. A component implementing a particular scheme may be replaced by another without affecting other components in an application. Moreover, it is also possible to link components that provide different schemes with an application, and to interactively select the scheme to achieve the optimum efficiency for a particular computation at runtime.



This chapter presents the object classes for sparse matrix manipulation developed in the present work. Section 12.2 describes pitfalls encountered in the implementation of sparse matrix manipulation components implemented in conventional procedural languages. Section 12.3 provides an overview of the classes for sparse matrices. Section 12.4 presents the design of the *Sparse* class that provides a standard interface for node-based sparse classes. Sections 12.5 and 12.6 describe two derived classes of the *Sparse* for two different sparse storage schemes. Finally, Section 12.7 describes a testing case for the two classes.

## 12.2 Sparse Matrix Manipulation in Procedural Languages

There are many implementations of sparse matrix manipulation packages designed following functional approaches and implemented in FORTRAN. The specification of a set of subroutines implementing the active column scheme (i.e., the skyline scheme) is chosen as an example and shown in Figure 12.1. This specification is summarized from a finite element program STAP listed in (Bathe, 1982). The parameter lists of the subroutines listed in this figure are modified by the author so that the communications among these subroutines and applications using these subroutines are only via parameter lists.

This set of subroutines is used in an application in the following steps:

1. The array *MHT* for the heights of columns in the sparse matrix is first allocated by the application using the approach described previously in Chapter 11. The subroutine *COLHT* is then called for each set of unknown connectivities (or elements) to calculate the column heights.
2. The array *MAXA* is allocated. The addresses of diagonal elements in the array *A* for the sparse matrix are calculated from *MHT*. The length of *A*, *NWK*, is also calculated.

```

SUBROUTINE COLHT(MHT, NEQ, LM, ND)
  -- to calculate column heights from unknown connectivity

SUBROUTINE ADDRESS(MAXA, NNM, MHT, NEQ, NWK)
  -- to calculate addresses of diagonal elements in the array
  -- storing the sparse matrix whose column heights are known

SUBROUTINE ADDBAN(A, NWA, MAXA, NNM, S, LM, ND)
  -- to assemble a sub-matrix into the sparse matrix

SUBROUTINE COLSOL(A, V, MAXA, NEQ, NWK, NNM, KKK)
  -- to decompose the sparse matrix and
  -- to solve a set of linear simultaneous equations

Formal parameters:
integer NEQ           number of unknowns of the linear equations system
integer NNM           NEQ+1
integer NWK           the size of the array storing the sparse matrix
integer ND            dimension of sub-matrices
integer MHT(NEQ)     an array storing active column heights
integer MAXA(NNM)    an array storing addresses of diagonal elements
integer LM(ND)       the unknown connectivity array
real A(NWK)           the array storing the sparse matrix
real V(NEQ)           the right-hand-side vector and the solution vector
real S(ND*(ND+1)/2)  the array storing sub-matrices
integer KKK           a flag for the subroutine COLSOL,
                     KKK = 1, for decomposition of the sparse matrix
                     KKK = 2, for solving the equations

```

Figure 12.1 Specification of a set subroutines implementing the skyline scheme

3. The array *A* is allocated for storing the sparse matrix and sub-matrices are assembled to the sparse matrix by calling the subroutine *ADDBAN*.
4. The subroutine *COLSOL* is called with *KKK* being 1 to decompose the sparse matrix.
5. The subroutine *COLSOL* is called with *KKK* being 2 to solve a set of linear simultaneous equations.

The problems involved with the use of this set of subroutines are apparent:

- The abstraction level is low in that the sparse matrix is represented not as a single variable but a set of arrays and variables. Programmers must allocate memory for each of the arrays explicitly and to pass these arrays and other variables to these subroutines in a correct order.
- No information about the sparse matrix can be hidden unless strict rules are enforced in the use of this set of subroutines.
- It is difficult if not impossible and meaningless to design a standard interface for components of this sort for different sparse storage schemes. This is because an application is responsible for the memory allocation of the arrays used by the component implementing a particular storage scheme.

A similar functional approach may also be implemented in the C language.

Nevertheless, a certain level of abstraction and a certain degree of information hiding may be achieved by the use of modularity in the FORTRAN and C languages. The object-oriented approach may also be followed in these languages to develop a sparse matrix manipulation component. Figure 12.2 shows an object-oriented design of a sparse matrix component implementing the skyline scheme in the C language.

```

typedef enum { created, realized, assembled, decomposed } SMstate;

typedef struct _Sparse {
    SMstate  s;      /* state of the sparse matrix */
    int      ec;     /* number of unknowns      */
    int*     colh;   /* height of each column   */
    double** a;     /* pointer to the matrix elements */
    int (*new_sparse)(Sparse sp, int n);
    int (*connectivity)(Sparse sp, int * uc);
    int (*realise)(Sparse sp);
    int (*assemble)(Sparse sp, double** sub_matrix, int* uc);
    int (*decompose)(Sparse sp);
    int (*solve)(Sparse sp, double* r, double* s);
} SparseRec, *Sparse;

```

**Functions contained in the structure:**

**new\_sparse:** to create and initialise an instance of the SparseRec structure  
**connectivity:** to calculate the column height from unknown connectivity  
**realise:** to allocate memory for the sparse matrix  
**assemble:** to assemble a sub-matrix to the sparse matrix  
**decompose:** to decompose the sparse matrix  
**solve:** to solve a set of linear simultaneous equations

**Parameters passed to these functions:**

<b>int n</b>	<b>number of unknowns</b>
<b>int* uc</b>	<b>a pointer to an unknown connectivity array</b>
<b>double** sub_matrix</b>	<b>a pointer to a sub-matrix</b>
<b>double* r</b>	<b>the right-hand-side vector</b>
<b>double* s</b>	<b>the solution vector</b>
<b>Sparse sp</b>	<b>pointer to a Sparse instance to be operated on</b>

**Figure 12.2** An object-oriented design of a sparse matrix component in C

The data structure shown in Fig. 12.2 defines a self-sufficient object class. The approach of self-sufficient objects was discussed previously in Chapter 4. The problem associated with this approach is that every instance of a class physically contains references to all functions that may be applied to it. This leads to memory overhead if many objects are created for an application. However, only a few number of sparse matrices are usually used in many applications. Thus, this design is reasonable for such applications.

The design listed in Figure 12.2 is definitely better than the functional design listed in Figure 12.1. A sparse matrix is represented as an instance of type *SparseRec*, and the necessary variables to represent the sparse matrix are bound together in the structure. Also, the structure contains a state variable, *s*, so the instance remembers its own state that can be used to direct operations on the instance. Applications are relieved from handling memory allocation for the sparse matrix performed here by functions contained in the structure. However, strict rules are still necessary to avoid applications to access the variables contained in the structure.

A standard interface for components implementing different sparse matrix storage schemes may also be achieved. However, this is done by following strict rules or disciplines in the design of these components rather than enforced by the compiler of the implementation language. Also, it is very difficult, if not impossible, to link functions for different storage schemes having a standard interface with an application in FORTRAN and C because a single name cannot be used for more than one function, i.e., function names cannot be overloaded. With the object-oriented methodology and object-oriented language, these problems can be well solved.

### 12.3 Overview of Sparse Matrix Classes

The abstract class facility of the C++ language is used in the design of the classes for sparse matrix manipulation. As discussed previously in Chapter 4, an abstract class is used to provide a framework and a standard interface for object classes representing a set of similar abstractions.

There are two types of abstractions of sparse matrices: unknown based and node based. In the first type, the unknowns of a linear simultaneous set of equation for which the coefficient matrix is a sparse matrix are represented as a vector of scalar variables. Entries in the sparse matrix are scalar elements.

In the second type, scalar unknowns are grouped into many sets consisting of several unknowns. These sets are referred to as nodes. The unknowns associated with a nodal point in a finite element analysis are an example of such a set. In the node based abstraction, unknowns of the linear simultaneous equations are represented as a vector of sub-vectors. The sparse matrix of the linear simultaneous equations is represented as a super-matrix because each of its elements is a sub-matrix. The node-based abstraction is more meaningful than the unknown-based one for some applications.

Both types of abstraction are implemented in the present work, and each corresponds to an abstract class. For a specific sparse storage scheme, a class can be derived from an appropriate abstract class. However, only the node-based classes are described here.

The abstract class representing the node-based sparse matrix abstraction is named *Sparse*. It specifies the standard interface for node-based sparse matrix abstractions. However, classes derived from this class do not necessarily have to be implemented based on sub-matrices. Currently, two derived classes of the *Sparse* class have been developed: the *ActiveColumn* class and the *SGraph* class.

The *ActiveColumn* class implements the active column storage scheme (Bathe, 1982). The *SGraph* class implements the graph-based storage scheme developed by Shi (1990). The most significant feature of this scheme is that the efficiency of solving linear simultaneous equations depends only slightly on the numbering of the nodes. This aspect will be discussed further in detail in Section 12.6.

#### 12.4 Design of the *Sparse* Class

A simplified version of the *Sparse* class declaration is listed in Figure 12.3, where the prototype of some member functions is not shown to save space. A complete listing may be found in (Zhang et al., 1990a).

##### 12.4.1 Representation of Sparse Matrices

*Sparse* is an abstract class. It represents the general sparse matrix characteristics to any particular sparse matrix storage scheme. It is used as a base class for classes implementing different sparse matrix storage schemes. No assumption is made about the storage scheme by this class, and the matrix can be symmetric or non-symmetric. Since *Sparse* is an abstract class, no object of *Sparse* can be created. However, objects of any of its derived classes can be represented as objects of the *Sparse* class. As a result, all such objects can be manipulated in the same way, with the exception of their creation. This aspects is demonstrated in the example application described in Section 12.7.

##### 12.4.2 States of *Sparse* Objects

An object of a class derived from the *Sparse* class is referred to as a *Sparse* object. A *Sparse* object can exist in one of the following five states after it is

An excerpt from the *Sparse* class declaration

```

class Sparse {
protected:
    static int dn;          // dimension of node
    static Real penalty;    // penalty factor to impose constraints
    static ErrorHandler eh; // error handler
    int nc;                // node count
    SMstate s;            // state of the linear system
    friend class SVector;
public:
    Sparse(int node_count, int node_dim = Sparse::dn);
    virtual ~Sparse() { }
    virtual int reset_node_count() = 0;
    virtual int size() = 0;
    virtual int copy(Sparse& rp) = 0;
    virtual int report_storage(ostream& os = cout) = 0;
    virtual int connection(IntVector& c) = 0;
    virtual int optimise() = 0;
    virtual int realise() = 0;
    virtual int assemble(Matrix& subm, IntVector& c) = 0;
    virtual int constraint(int n, RealVector& v, Real p=Sparse::penalty) = 0;
    virtual int disable_node(int id) = 0;
    virtual int disable_unknown(int id, IntVector& v) = 0;
    virtual int decompose(Real tol = (Real) 0.0001) = 0;
    virtual int solve(RealVector& rh, RealVector& rs) = 0;
    virtual Real determinant() = 0;
    virtual int clean() = 0;
    virtual int assign(Sparse& rp) = 0;
    virtual int plus_assign(Sparse& rp) = 0;
    virtual int minus_assign(Sparse& rp) = 0;
    virtual int multiply_rv(RealVector& rv, RealVector& rs) = 0;
    virtual int multiply_lv(RealVector& lv, RealVector& rs) = 0;
    virtual Real triple_product(RealVector& lv, RealVector& rv) = 0;
    virtual int print(ostream& os = cout) = 0;
    virtual int read(istream& is = cin) = 0;
};

```

Figure 12.3 A simplified version of the *Sparse* class declaration



created: created, optimized, realized, assembled, and decomposed.

**The Created State:** When a *Sparse* object is created, sufficient memory is allocated for storing its properties except the memory for storing coefficients of the sparse matrix the *Sparse* object represents. A *Sparse* object is in a created state until it is realized. The connectivities between the nodes determine the non-zero coefficients of the coefficient matrix. These connectivities are specified to a *Sparse* object when it is placed in the created state.

**The Optimized States:** The optimized state is optional and indicates that the node numbering of a system represented by the *Sparse* object has been optimized.

**The Realized State:** The memory for non-zero coefficients including any coefficients that are made non-zero during the matrix decomposition process is allocated when a *Sparse* object is realized. Sub-matrices can be assembled to the matrix represented by a *Sparse* object only after the object is realized.

**The Assembled State:** A *Sparse* object is in an assembled state if at least one sub-matrix has been assembled to the matrix that it represents. A *Sparse* object can be decomposed only when it is in an assembled state.

**The Decomposed State:** A *Sparse* is in a decomposed state if the matrix it represents has been decomposed. A decomposed *Sparse* object is ready to be used for solving linear simultaneous equations and for calculating the determinant of the matrix.

### 12.4.3 Properties of the *Sparse* Class

The interface defined by the *Sparse* class is based on nodes. A node contains  $q$  unknowns, where  $q$  is greater than or equal to 1. The declaration of the *Sparse* class contains member variables for representation of general sparse matrices. As shown in Figure 12.3, these variables include: the number of unknowns per-node "*dn*", and the number of nodes "*nc*", the penalty factor "*penalty*" used to impose constraints, the member object "*eh*" for exception-handling, and the state variable "*s*" which is of the enumeration type *SMstate*. The number of unknowns per-node is fixed for a particular application.

### 12.4.4 Classes Used with the *Sparse* Class

Several classes are used with the *Sparse* class and referred to in the *Sparse* class declaration. These classes are:

- *Matrix*: described in Chapter 11. The matrices from which a sparse matrix is assembled are represented as objects of the *Matrix* class.
- *IntVector*: defined by using the parameterized vector class, *Vector(T)*, discussed in Chapter 10. This class represents vectors of integers and it serves as the class of the node connectivity arrays.
- *SVector*: a derived class of the *RealVector* class. This class serves the following purposes: (1) it creates *RealVector* objects of dimension equal to the total number of unknowns based on the number of nodes, (2) it assembles sub-vectors to a *RealVector* object, and (3) it retrieves sub-vectors from a *RealVector*.

The type of sparse matrix elements is *Real*, can be either *float* or *double*, and is determined at compilation time.

### 12.4.5 Interface of the Sparse Classes

The function prototypes contained in the declaration of the *Sparse* class specify the standard interface for sparse matrix classes. The *Sparse* class implements only a few of these functions itself. These functions are mostly for retrieving and setting values of common member variables defined in the class. Most of these functions are *pure virtual* functions. A pure virtual function of a class is the one that must be implemented by each of its derived classes. The derived classes of *Sparse* implement the majority the interface.

#### 12.4.5.1 Object Creation and Destruction

A *Sparse* object can be created statically by defining an object of a derived class or dynamically by using the operator *new*. When a *Sparse* object is created, the number of nodes and the number of unknowns per node (optional) are specified. The number of unknowns per node has a default value of two which can be reset through a member function of the *SVector* class. When a *Sparse* object is created, the number of unknowns per node of the *SVector* class is set to the same value as the *Sparse* class. Thus, in an application, one or more *Sparse* objects can be created. However, these objects must have the same number of unknowns per node.

#### 12.4.5.2 Establishing the Nodal Connectivity and Realization

The connectivity between nodes in a system, which determines the storage of a *Sparse* object, is established by passing the nodal connectivity array to the *Sparse* object using the member function *Derived::connectivity*. Here, the name "*Derived*" refers to the name of a derived class. Specifying the class name with the member function name attempts to emphasize that this function is

implemented by a derived class. A nodal connectivity array contains a set of node numbers. Listing two node numbers  $i$  and  $j$  in a connectivity array indicates that the sub-matrices  $(i, j)$ ,  $(j, i)$ ,  $(i, i)$  and  $(j, j)$  in the sparse matrix are non-zero.

A node or an unknown of a node can be disabled such that the node or unknown will not be included in storage of the linear simultaneous equations. A unknown also can be restricted such that the solution of the linear equation system for this unknown is zero or a specified value.

The order of the nodal numbering can be optimized to reduce the memory used to store the sparse matrix. This is done by using the method *Derived::optimize*. A *Sparse* object is realized by using the member function *Derived::realize*. Memory for the sparse matrix is allocated when the object is realized.

#### 12.4.5.3 Solving Linear Simultaneous Equations

The sparse matrix represented by a *Sparse* object is assembled by using the member function *Derived::assemble*. The matrix is decomposed into the form  $LU$  for unsymmetric matrices or  $LDL^T$  for symmetric matrices by using the member function *Derived::decompose*. A decomposed sparse matrix object is ready to be used for solving linear simultaneous equations by using the member function *Derived::solve*. Two *RealVector* objects are passed to this function. One holds the right-hand-side vector, and the other holds the solution vector. Invoking this function through a *Sparse* object that has not been decomposed will also cause the object to be decomposed. The determinant of the sparse matrix is obtained by using the member function *Derived::determinant*.

#### 12.4.5.4 Other Operations for *Sparse* Objects

Other operations for sparse matrix manipulations include sparse matrix assignment, multiplication of a sparse matrix with a vector, and input/output of a sparse matrix. Because a *Sparse* object remembers its state, the member functions for multiplying the sparse matrix with a vector are performed correctly even after the matrix is decomposed.

### 12.5 Design of the *ActiveColumn* Class

The *ActiveColumn* class is derived from the abstract class *Sparse*. The *ActiveColumn* class implements the active column (skyline) storage scheme. A simplified version of the *ActiveColumn* class declaration is listed in Figure 12.4, where only some prototypes of member functions are listed for compactness. A complete listing of the declaration may be found in (Zhang et al., 1990a).

The coefficients in the upper triangle of a symmetric sparse matrix are stored in a two-dimensional array represented by a member variable *a* which is a pointer to pointers to the *Real* type. Each column of the skyline in the upper triangle of the matrix is stored in a row of the array. Each row of the array may have different number of entries. Two profiles, *pn* and *pf*, are used to record respectively the number of sub-matrices and the number of elements in each row. Both profiles are member objects of the *IntVector* class. *pn* is the profile in terms of the nodes, and *pf* is the profile in terms of the unknowns.

The decomposed form of the sparse matrix,  $LDL^T$ , is stored in the same storage as the undecomposed matrix. The elements of the diagonal matrix *D* are stored separately in an array represented by a pointer to the type *Real*. This pointer is the member variable *d*.

The *ActiveColumn* class implements the interface specified in the abstract class *Sparse* as described in the previous section.

An excerpt from the *Sparse* class declaration

```

class ActiveColumn : public Sparse {
private:
    int    ec; // dimension of the matrix
    IntVector pn; // profile in terms of node
    IntVector pf; // profile in terms of unknown
    Real**  a; // matrix coefficients
    Real*   d; // elements of the diagonal factor matrix D
public:
    ActiveColumn(int count, int dim = Sparse::dn) :
        Sparse(count, dim), pn(count), pf(count*dim)
        {   ec = count * dim; s = created;   }
    ~ActiveColumn() { if (is_realized()) free_memory(); }
    int connection(IntVector& c);
    int optimise();
    int realise();
    int assemble(Matrix& subm, IntVector& c);
    int decompose(Real tol = (Real) 0.0001);
    int solve(RealVector& rh, RealVector& rs);
    Real determinant();
};

```

Figure 12.4 A simplified version of the *ActiveColumn* class declaration

## 12.6 Design of the *SGraph* Class

The graph-based scheme is an efficient direct-solution method for linear simultaneous equations (Shi, 1990). It has a node renumbering algorithm as one of its inherent components. This leads to one of its distinguishing features whose efficiency depends only slightly on the node numbering of a system. This feature is significant for problems where it is difficult or computationally expensive to optimize the node numbering for the skyline scheme.

Although this method is not limited to systems with a symmetric sparse matrix, only the graph-based scheme for symmetric sparse matrices is implemented in the present work. A detailed description of the graph-based scheme can be found in (Shi, 1990). The basic ideas of the method are described

briefly in Section 12.6.1. The major steps in the node renumbering algorithm are summarized in Section 12.6.2. The formulation of the matrix decomposition for a matrix with sub-matrices as its entries is listed in Section 12.6.3. Section 12.6.4 discusses the implementation of the method.

### 12.6.1 The Graph-Based Sparse Storage Scheme

Most existing sparse matrix schemes are compact methods in that their efficiency depends on the band-width of the sparse matrix. A number of efforts have been spent on optimization algorithms to reorder node numbers and minimize the band-width. The graph-based method reorders the node numbers to minimize the number of non-zero entries produced during the matrix decomposition. The resulting non-zero entries of the sparse matrix before and after decomposition may not be necessarily located close to the diagonal. Rather, they may be distributed over the entire matrix. These non-zero entries are recorded during the nodal renumbering. Memory is allocated only to store the non-zero entries. Because memory is also required to store information about the distribution of non-zero entries, this method is most efficient when the entries of a sparse matrix are matrices instead of scalars.

Nodal connectivities or the topological relationships of a system correspond to a graph. Each node of the system is a node in the graph. The connectivity between two nodes of the system corresponds to a line connecting the two corresponding nodes in the graph. The distribution of non-zero entries in a sparse matrix before decomposition is represented by a graph. Eliminating a row in the Gaussian elimination process, or correspondingly decomposing a row in the matrix decomposition process, will erase the corresponding node from the graph. Eliminating a row in the Gaussian elimination process may generate new non-zero entries in the matrix. Erasing a node from the graph may also produce new lines according to certain rules to be discussed in the next section. Thus, a

one-to-one correspondence can be established between the Gaussian elimination process (i.e., the matrix decomposition process) and a graph manipulation process.

In the implementation of the graph-based scheme, the graph is represented by a software object. The initial graph is constructed according to the nodal connectivities specified for a given system. The row elimination process is then simulated by operations on the graph along with a nodal renumbering process to reduce the number of non-zero entries produced. The results of such a simulation are a new nodal numbering and a graph based on the new node numbering. This graph records the distribution of non-zero entries including those non-zero entries produced in the sparse matrix during the matrix decomposition. The major steps of the simulation process will be discussed in the next section.

### 12.6.2 Determining New Nodal Numbering

The graph can be represented as a two-dimensional array referred to as  $G$  for description purpose. The array  $G$  initially has  $N$  rows representing an  $N$ -node graph. In the  $i$ 'th row of the array, the numbers of the nodes that connect to the  $i$ 'th node by a line in the graph are recorded. Thus, if two nodes numbered respectively as  $i$  and  $j$  are connected in the graph, the number  $i$  is recorded in the  $j$ 'th row, and  $j$  is recorded in the  $i$ 'th row of the array  $G$ .

The simulation of the matrix decomposition process and the determination of the new node numbering is performed in the following two major steps.

1. **Finding and Erasing the Least Connected Node:** The node-renumbering process starts by searching the array  $G$  for the node having the least connections with other nodes. This node is chosen as the first node in the new node numbering. If the node is node  $i$ , the  $i$ 'th row and the  $i$ 'th



column of the sparse matrix are moved to the first row and the first column respectively. The new first row can then be eliminated or decomposed. This corresponds to the erasing of node  $i$  from the graph or the removal of the node number  $i$  from all rows of the array  $G$ .

2. Determining the Generated Line: If the  $i$ 'th row of the array  $G$  is not empty, a new line is generated between any two nodes in the  $i$ 'th row if the two nodes are not already connected in the graph. These new connectivities are then recorded in the array  $G$ . A generated connectivity between two nodes, say node  $j$  and node  $k$ , indicates that two non-zero entries,  $(j, k)$  and  $(k, j)$ , are produced during the elimination of the node  $i$ . Finally, the  $i$ 'th row of the array  $G$  is marked such that it will not be processed again.

The operations described above on the  $N$ -node graph result in a new graph having  $(N-1)$  nodes. The new graph is still represented by the array  $G$ . These operations are then repeated to operate on the new  $(N-1)$ -node graph. The second node under the new nodal numbering is then found, and a new  $(N-2)$ -node graph is produced. This process is repeated for  $(N-1)$  times. Also, a graph is produced during the process to represent the distribution of non-zero entries of the decomposed matrix associated with the new node numbering.

Memory is allocated for all lower triangle and diagonal non-zero entries of the sparse matrix according to the graph under the new nodal numbering. The sparse matrix can then be assembled.

In general, the initial nodal numbering of a system affects slightly the number of non-zero sub-matrices to be stored. For some problems, a unique number of sub-matrices can be obtained independent of the nodal numbering. However, for other problems, different numbers of sub-matrices may be obtained from different initial nodal numberings for a given system. The differences are usually not significant.

### 12.6.3 Formulation of Matrix Decomposition

The formulation for the decomposition of a matrix whose entries are sub-matrices is the same as the formulation for common matrix decomposition. This formulation can be simply expressed as

$$A = LDL^T$$

A is a N by N symmetric matrix. The entries of A,  $A_{ij}$ , are sub-matrices of dimension q by q where q is the number of unknowns per node. L is a lower-triangle unit matrix. Its entries,  $L_{ij}$ , are sub-matrices of dimension q by q, and its diagonal entries are unit matrices. The matrix D is a diagonal matrix, and each of its diagonal entries,  $D_{ij}$ , is a q by q symmetric matrix. The formula for determining the matrices L and D are as below

$$L_{ij}D_{jj} = A_{ij} - \sum_{k=1}^{j-1} L_{ik}D_{kk}L_{jk}^T \quad (j < i)$$

and

$$D_{ii} = A_{ii} - \sum_{k=1}^{i-1} L_{ik}D_{kk}L_{ik}^T$$

### 12.6.4 Implementation

This section discusses the issues involving the implementation of the graph-based scheme. This scheme is implemented in a class *SGraph* derived from the *Sparse* abstract class. The implementation of the *SGraph* class is based on the original C code written by Dr. Gen-Hua Shi. Several classes are internally defined and used by the *SGraph* class. These classes include: *SubMatrix* used for the representation of a sub-matrix entry in the sparse matrix; *IndexedMatrix* used for an entry in the graph array G; and *NodeTable* used for a row of the graph array G. These classes and the *SGraph* class are described below.

#### 12.6.4.1 Representation of Sub-Matrices

The *SubMatrix* class is defined to represent a group of square matrices having the same dimension. Entries of a sparse matrix are matrices of this sort. The *SubMatrix* class is derived from the *Matrix* class described previously in Chapter 11. Member functions are defined for manipulating this specific matrix type.

#### 12.6.4.2 Representation of Entries in the Graph Array

The *IndexedMatrix* class is defined to represent entries of the graph array G. An entry of the array G contains a node number as well as a sub-matrix if the entry corresponds to a lower-triangle entry in the sparse matrix.

#### 12.6.4.3 Representation of Rows in the Graph Array

The number of entries in a row of the graph array cannot be determined in advance, and the number may increase during the graph operation. Thus, a class *NodeArray* representing arrays of *IndexedMatrix* objects is defined first based on a parameterized array class. This parameterized array class is an older version of the *ExtArray(T)* class described in Chapter 10.

The *NodeTable* class is defined using the *NodeArray* class as a base class. Member functions are defined in the *NodeTable* class to insert an entry to a row of the graph array G, delete an entry from a row, lookup an entry from a row, and sort entries according node numbers etc. This class also contains a *SubMatrix* object storing the inverse of an entry sub-matrix of the matrix D. Thus, the entries of both the matrix D and its inverse  $D^{-1}$  are stored after a sparse matrix is decomposed for efficiency. Moreover, when a sparse matrix is

decomposed, the sub-matrix  $L_{ij}$  ( $j < i$ ) is stored in the same storage as the sub-matrix  $A_{ij}$ , and the sub-matrix  $D_{ii}$  is stored in the same storage as  $A_{ii}$ . A simplified version of the *NodeTable* class declaration is shown in Figure 12.5.

#### 12.6.4.4 Representation of the Graph

The graph array  $G$  is represented by the class *SGraph*. A simplified version of the *SGraph* class declaration, where only some of the member function prototypes are shown for conciseness, is listed in Figure 12.6. A complete listing is given in (Zhang et al., 1990a).

The *SGraph* class contains the following member variables:

- The number of effective nodes "*nnc*". An effective node is the one that connects with at least one other node and that is not disabled.
- The graph array "*graph*". It is an array of *NodeTable* objects, but declared as a variable of type `void**` to hide the internal classes such as *NodeTable* and *IndexedMatrix* from the application.
- A pointer to an *IntVector* object "*pnew*". This *IntVector* object stores the new node numbers with the old node numbers as indices.
- A pointer to an *IntVector* object "*pold*". This *IntVector* object stores the old node numbers with the new node numbers as indices.
- A real variable "*det*" for the determinant of the sparse matrix. The determinant of a sparse matrix is calculated when the matrix is decomposed.

The *SGraph* class contains member functions implementing the interface specified in the abstract class *Sparse* described in Section 12.4.

An excerpt from the `NodeTable` class declaration

```

class NodeTable : private NodeArray {
    static int  init_size;    // initial size of the NodeArray
    static int  size_incr;   // size increment of the NodeArray
    int        ndc;         // number of node in the table
    int        diagnl;      // index of the diagonal sub-matrix
    SubMatrix* pdi;        // pointer to the inverse of the diagonal
public:
    NodeTable(int d) : (NodeTable::init_size, NodeTable::size_incr)
                      { ndc = 1;  elem(diagnl = 0).id = d;
                      pdi = (SubMatrix*) 0; }
    NodeTable(NodeTable& x) { ndc = x.ndc;  diagnl = x.diagnl;
                          pdi = x.pdi; }
    ~NodeTable();
    Node& operator[](int i)
        { if (i < 0 || i >= ndc)
          eh.error(ary_index,
                  "NodeTable index out of range");
          return NodeArray::elem(i); }
    Node& elem(int i) { return NodeArray::elem(i); }
    int insert(int c); // insert a node before the diagonal
    int append(int c); // append a node at the end
    int move_forward(int c); // move a node behind diagonal to ahead
    int move_backward(int c); // move a node ahead diagonal behind
    int lookup(int c, Node& node); // lookup a node identified by id (c)
    void allocate(); // allocate the sub-matrices
    void deallocate(); // deallocate the sub-matrices
    void erase(int c); // delete a node from the table
    void erase(); // delete all nodes except the diagonal
    void clean(); // zero all sub-matrices
    void renumber(IntVector& t); // renumbering id for all nodes
    void sort(); // sort such that id in increasing order
    void report_connectivity(ostream& os = cout);
    void print(ostream& os = cout);
};

```

Figure 12.5 A simplified version of the `NodeTable` class declaration

An excerpt from the *SGraph* class declaration

```

class SGraph : public Sparse {
private:
    int    nnc;    // number of effective nodes
    void** graph; // the graph
    IntVector* pnew; // array of new node number with old as index
    IntVector* pold; // array of old node number with new as index
    Real det;    // the determinant
public:
    SGraph(int count, int dim = Sparse::dn);
    ~SGraph();
    int connection(IntVector& c);
    int optimise();
    int realise();
    int assemble(Matrix& subm, IntVector& c);
    int decompose(Real tol = (Real) 0.0001);
    int solve(RealVector& rh, RealVector& rs);
    Real determinant();
};

```

Figure 12.6 A simplified version of the *SGraph* class declaration

## 12.7 Testing of Sparse Matrix Classes

A test program for the sparse matrix object classes is described in this section. A rectangular region is discretized into  $(m+1) \times (n+1)$  nodes and  $n \times m$  elements as shown in Figure 12.7. Each node is assumed to have 6 unknowns. This problem does not have any physical meaning and is only used for testing of sparse matrix classes. The order of node numbering is fixed as shown in the figure. The node numbers increase from left to right and from top to bottom. An excerpt of the testing program is shown in Figure 12.8. A complete listing of the program may be found in (Zhang et al., 1990a).

There are two functions in the test program, a void function *make\_matrix* and the *main* function. In the main function, the input data includes the

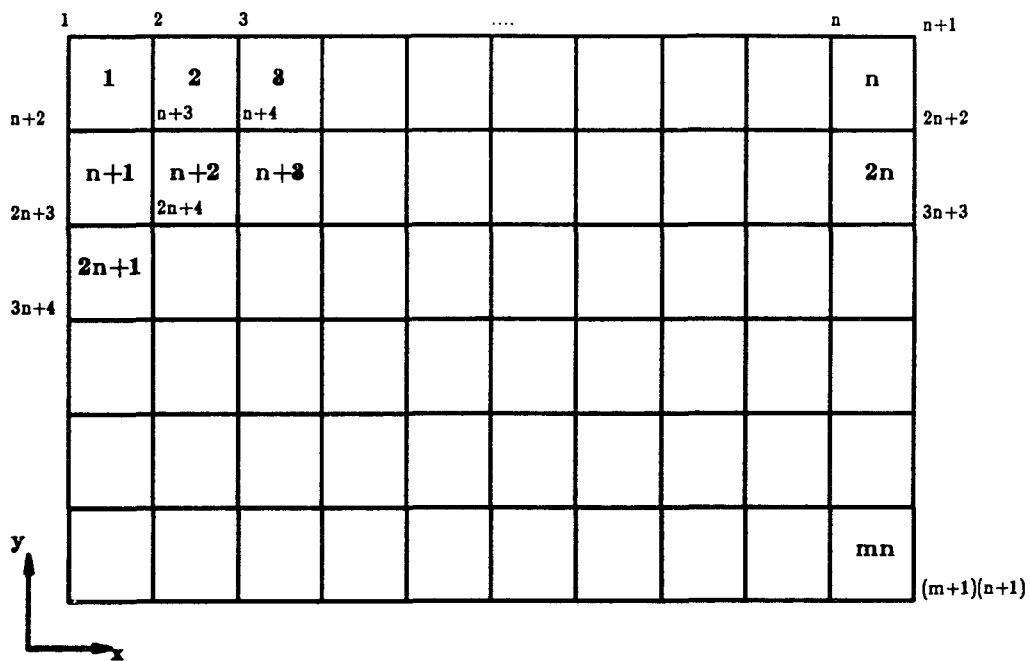


Figure 12.7 The mesh of the testing case

```

#include "actvcmCC.h" // include the ActiveColumn class declaration
#include "sgraphCC.h" // include the SGraph class declaration

Sparse* psp; // the pointer to the Sparse object
int elem_x, elem_y; // number of elements in the x-direction and in the y-direction
int dim; // number of unknowns per node

void make_matrix()
{
    This function generates the sparse matrix. It uses the function
    (*psp).connection to establish the node connectivity, uses the
    function (*psp).realize to realize the object, and uses the
    function (*psp).assemble to assemble the sparse matrix.
}

main()
{
    int nd_count; // number of nodes
    int m_type; // type of sparse matrix to be used
    cin >> elem_x >> elem_y >> dim >> m_type; // input parameters

    SVector::set_dn(dim); // set the number of unknowns per node
    nd_count = (elem_x + 1) * (elem_y + 1);
    if (0 == m_type)
        psp = new ActiveColumn(nd_count); // create an ActiveColumn objet
    else
        psp = new SGraph(nd_count); // create a SGraph objet
    make_matrix(); // generate the sparse matrix
    (*psp).decompose(); // decompose the sparse matrix
    SVector v1(nd_count), s1(nd_count), s2(nd_count);
    for (int i = 0; i < nd_count*dim; ++i) // generate right-hand-side vector
        v1(i) = double(i) + 1.0;
    (*psp).solve(v1, s1); // solve simultaneous equations
    s1.print(); // print the solution
}

```

Figure 12.8 An excerpt of the testing program for *Sparse* classes



number of elements in the x- and y-direction, the number of unknowns per node, and a flag indicating the storage scheme to be used. A sparse matrix object is created based on the input data. The function *make\_matrix* is then called to establish the nodal connectivities of the sparse object, to realize, and to assemble the sparse object. The stiffness matrix for each element is generated artificially.

After the sparse matrix object is assembled, it is then decomposed and used to solve a set of linear simultaneous equations. The advantage of the standard interface feature of sparse classes is demonstrated in this testing program. In the function *make\_matrix*, the sparse matrix object is manipulated without knowing the actual class of the object. In the *main* function, the actual class of the sparse matrix is checked only in the statement where the sparse matrix object is created.

A comparison of the efficiency of different sparse matrix classes has been performed on a Sun 3/60 workstation. The intent here is to provide only a rough estimate of the efficiency of the *ActiveColumn* and *SGraph* classes, rather than to make a comprehensive comparison between these two schemes. For a given number of nodes  $n \times m = 480$ , six cases are computed for different combinations of  $n$  and  $m$ . Table 12.1 shows the storage and CPU times used by the two sparse classes for the six cases.

The six cases are listed in the table in the order of increasing band-width of the sparse matrix. The efficiency of the *ActiveColumn* class is greater for the smaller band-widths as expected. Conversely, the efficiency of the *SGraph* class shows little dependence on the nodal numbering. The memory and CPU time used in the case of  $n = 40$  and  $m = 12$  is very close to those used in the case of  $n = 12$  and  $m = 40$ . In fact, these two cases correspond to the same mesh but with different orders of nodal numbering. However, the memory and CPU time used for the two cases  $n = 20$ ,  $m = 24$ , and  $n = 24$ ,  $m = 20$  shows a clear

Table 12.1 Comparison of efficiencies of *Sparse* classes

n	m	Nodes	Unknowns	ActiveColumn		SGraph	
				Storage used	CPU (sec)	Storage used	CPU (sec)
40	12	533	3198	759303	1036	313632	356
30	16	527	3162	586125	629	335340	423
24	20	525	3150	482319	432	328140	400
20	24	525	3150	413199	323	347724	469
16	30	527	3162	344205	228	343728	455
12	40	533	3198	275463	151	310212	347

Notes: The storage used is estimated in unit of double precision numbers.

difference, even though these two cases also correspond to the same mesh.

It can also be noted from the table that for the cases where a lower band-width can be easily achieved by numbering the nodes properly, the *ActiveColumn* class is faster and uses less memory than the *SGraph* class. However, for cases where an optimum band-width can not be easily achieved, the *SGraph* class may be more efficient.

The code for the active column scheme can be optimized more easily than the graph-based scheme. The current implementation of the *SGraph* class uses the *Matrix* class to represent its sub-matrices. For problems having a lower number of unknowns per node such as the plane stress/strain finite element analysis, special matrix classes should be developed and used to represent sub-matrices for the *SGraph* class and to optimize the performance of the method. Moreover, the current implementation of the node renumbering strategy described in Section 12.6.2 may be improved to reduce further the node numbering dependence of the *SGraph* class.

## LIST OF REFERENCES

1. Bathe, K.-J., "Finite Element Procedures in Engineering Analysis", Prentice-Hall, New Jersey, 1982, 735pp.
2. Baugh Jr., J. W., and Rehak, D. R., "Object-Oriented Design of Finite Element Programs", in Computer Utilization in Structural Engineering, Proceeding of the sessions related to computer utilization at Structure Congress'89, San Francisco, CA., May 1-5, 1989, Edited by James K. Nelson, Jr, pp.91-100.
3. Duff, C. and Howard, B., "Migration Patterns", BYTE, October 1990, pp. 223-232.
4. Eckel, B., "Using C++", McGraw-Hill, New York, 1989, 617pp.
5. Fenves, G.L., "Object-Oriented Programming for Engineering Software Development", Engineering with Computers, No.6, 1990, pp.1-15.
6. Landers, J.A., "A Software Development Specification for Nonlinear Structural Analysis", Contract Report, CR 90.016, Department of Civil Engineering, University of California, Berkeley, July 1990, 18pp.
7. Lee, H.-H., "Engineering Design Optimization Capability Using Object-Oriented Programming Method with Database Management System", Ph.D. Dissertation, The University of Iowa, May 1989, 188pp.
8. Miller, G.R., "A LISP-Based Object-Oriented Approach to Structural Analysis", Engineering with Computers, No.4, 1988, pp.197-203.
9. Shi, G.-H., "Block System Modeling by Discontinuous Deformation Analysis", Computational Mechanics Publication, U.K., 1990.
10. Stroustrup, B., "The C++ Programming Language", Addison-Wesley, Reading, Massachusetts, 1987, 328pp.
11. Zhang, H., White, D. W., and Chen, W. F., "An Object-Oriented Matrix Manipulating Library", Structural Engineering Report, CE-STR-90-10, School of Civil Engineering, Purdue University, West Lafayette, Indiana, June 1990a, 77pp.

12. Zhang, H., White, D. W., and Chen, W. F., "The SESDE Object Library Manual", Structural Engineering Report, CE-STR-90-33, School of Civil Engineering, Purdue University, West Lafayette, Indiana, November 1990b, 108pp.

## CHAPTER 13 DATABASE MANAGEMENT SYSTEM REQUIREMENTS

This chapter gives a brief review on the current technology of Database Management Systems (DBMSs) and discusses an approach to integrating a commercial DBMS with the SESDE. The organization of this chapter is as follows. The importance of a database management system for computational software is introduced in Section 13.1. The basic requirements for a general database management system are described in Section 13.2. The older generations of database management technology including flat-file management, hierarchical and network database management, and relational database management, as well as the problems associated with these older technologies are described in Sections 13.3. The state-of-the-art of object-oriented database management is discussed in Section 13.4. The characteristics of engineering data and engineering computation environments are discussed in Section 13.5, and a brief overview of the state-of-the-art of engineering database management is given in Section 13.6. Finally, Section 13.7 discusses issues associated with the integration a commercial DBMS with the SESDE.

### 13.1 The Need for Database Management Systems

The purpose of database management systems is to maintain databases stored in the permanent or secondary storage (disks) of a computer system, and to transfer data from user to the program, or between code units in the program, or between different programs. A database is a collection of logically related persistent data that are produced, utilized, and shared by one or more programs. Persistent data are those that have a longer life than the programs

that produce and manipulate them. Figure 13.1 shows the basic relationship among an application, a database, and the database management system.

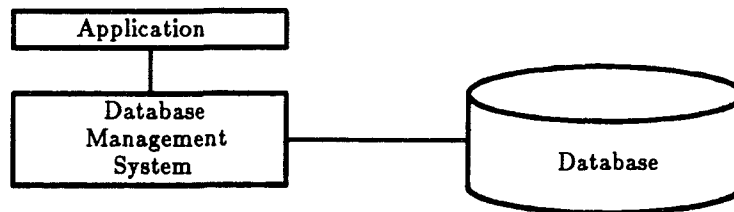


Figure 13.1 Basic relationship: an application with a database

Database management technology has been well developed and utilized in business data processing such as marketing and banking. Also, the importance of this technology for engineering applications in supporting data-intensive engineering applications software and integrated engineering software systems has been well recognized in recent years.

Engineering applications software often requires manipulation of a large amount of data for certain engineering tasks. Computer-Aided-Design (CAD) software is a typical example of this sort. A typical application needs to obtain input data either interactively or from databases in the computer file system. It must also perform validation checking of the input data to ensure the correctness of the computations. The code that handles the data input and validation checking is the most cumbersome and error-prone part in many programs.

Furthermore, in most modern applications, the input process is substantially compressed by the use of sophisticated user-interfaces and computer graphics. The resulting input data is then greatly expanded prior to or during the performance of the engineering tasks. The program also needs to

store the processing results in databases for further processing, which, in turn, become the input data of either the same program in the next execution or other programs which are integrated with this program.

Modern engineering software systems often integrate several programs due to the complex nature of engineering activities. These programs share common databases. A set of programs for structural design, analysis, and drafting is a typical example of integrated software systems. A database management system is a critical component for such a system in facilitating the communications and enforcing data consistency between these programs. Figure 13.2 shows a typical configuration of an integrated engineering software system.

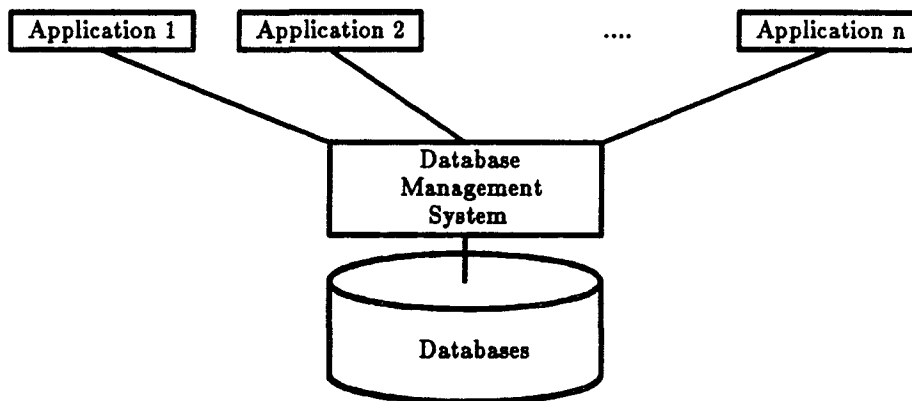


Figure 13.2 Typical configuration of an integrated system

Moreover, an object-oriented program also tends to be decentralized in that the program consists of a set of object classes and/or object sub-systems. The communication between objects and object sub-systems is accomplished by objects sending messages to each other. In order to do so, a message sender object must know the message receiver object. An example of this is that in a finite element analysis program, an element object must know the global

stiffness matrix object in order to assemble its element stiffness matrix to the global matrix. To prompt the independency between these classes and subsystems and the flexibility for modification and extensibility, the use of objects with global scope is discouraged. Thus, a database management system is also critical to transfer objects, either in the primary memory or in the secondary memory, between code units upon requests in such programs during the program execution.

In the absence of a formal database management system, a traditional application program owns its own set of files for storing persistent data, and each of these files has its own logical and physical structure. Persistent data, which are usually of a basic data type supported by programming languages such as integer, real, and text strings, are grouped into records, and each data item in a record is referred to as a field of the record. Records are stored in database files in a certain order. The logical structure of a database refers to the type, format and sequence of data fields packed in records as well as the relationships between these records. The physical structure refers to the physical sequence of the records in database files. If programs and the stored data depend heavily on each other, both programs and databases will be difficult to change.

When different programs in an integrated system need to access the same data, either a strict agreement has to be made among these programs about the logical and physical structure of the shared database files, or the shared data are stored in different database files according to the needs of each individual program. The second approach results in the shared persistent data being stored redundantly and causes difficulty in keeping the data consistency among different programs in an integrated system. Also, each program has to perform its own persistent data validation and manipulation resulting in duplication of code across these programs. Since programs heavily depend on each other, a modification of a single program could have a profound and unexpected effect on



many other programs. This results in difficulties in the extension of a system.

The problem with traditional data management via flat files is that the databases are mainly defined or viewed as the properties of each individual program, rather than independent and shared resources. To make databases shared and better managed, the most important principle that should be followed is data-independence. That is, a database should stand on its own and not depend on any particular application. To this end, DBMSs are needed to stand between applications software and their supporting databases and connect applications with databases.

DBMSs provide a uniform view on the data stored in databases and data access for applications according to the view to facilitate data-independence. With the assistance of a proper DBMS, applications need only care about which persistent data to be stored or retrieved rather than how to store and retrieve. Thus, database management technology has the capability to improve the flexibility for modification, extensibility, and standardization of applications software to a large extent.

During the last 25 years, database management technology has evolved through three full generations: flat-file management systems, hierarchical and network database management systems, and relational database management systems. It is now entering the fourth generation with object-oriented technology (Loomis, 1990). However, it is interesting to note that the successive generations have not completely replaced their predecessors. Rather, the older technology continues to exist along with the new methods. This is particular true in engineering application areas.

Each generation of database management technology has its own distinct data model. A data model is a style of describing and manipulating data in a database. Data models differ in the style in which data objects and relationships between data objects are described. They also differ in the manner in which

constraints on data objects and operations upon the database are expressed. Data models are used to describe schemas, which in turn describe specific collections of data in the database.

### 13.2 Basic Requirements

A common set of basic requirements for general database management systems is listed below.

- a. **Provision of Data Persistence.** This is the major role of a DBMS. A database exists outside the scope of any particular program run-unit. The data are stored in non-volatile storage and continue to exist even after the execution of the program that created them has terminated.
- b. **Provision of Program-Data Independence.** Program-data independence means that the physical placement and formatting of data, and the technique used to access the data in the database are hidden from applications which utilize the database. A database management system is said to provide program-data independence if it does not require modification of applications when a database is changed in its logical or physical structures. Several levels of data independence can be achieved by a DBMS. These levels may be classified based on the following changes required in the application when an associated database is changed:

- modification of statements,
- recompilation,
- relinking,
- nothing required.

An execution cost is associated with data independence. With computer processing cost decreasing and software maintenance cost increasing, data

independence provides cost savings by reducing costly program maintenance and extension at a small cost in execution performance.

- c. **Support of Data Semantics.** A DBMS should understand and handle certain data semantics. Semantics can be described in terms of what can and cannot be done with the data. The DBMS should have the capability to perform data validation checking. An application should describe what to be retrieved and stored according to the data model, and the DBMS should determine how the data can be accessed efficiently in the database.
- d. **Provision of multiple views on the same set of data.** A view is the capability to limit the visibility of data objects. A database may be viewed differently by different applications, and each view represents a certain aspect of the database. The underlying concept is information hiding. Views give applications selective access to a database such that only the data necessary for an application is accessed by the application. Views also support data-independence by providing a stable interface to the database even though the logical structure of the database might change. A view can be defined by hiding named fields in the logical structure of the database or by defining new fields which are not stored explicitly, but rather are derived or computed from stored hidden fields.
- e. **Primary storage management.** A DBMS should provide efficient ways to transfer data objects either in primary or secondary memory between code units upon requests. This is to prompt software flexibility, extensibility and standardization.
- f. **Secondary storage management.** A DBMS should provide efficient ways to represent and access both large-scale data objects and large collections of small data objects.

- g. Concurrency. This is a fundamental capability to a DBMS to support sharing of a single database.
- h. Recovery. This feature enables the DBMS to cope with system failures and to protect database contents from destruction. Failures can come from the processor, the network, the system software, the application software, and hardware.
- i. Ad hoc query facility. This enables users to access database contents without writing a program.

### 13.3 Older Database Management Technologies

This section describes first three older generations of database management technologies including file management, hierarchical and network database management, and relational database management technologies. Problems associated with these older technologies for engineering software are then discussed.

#### 13.3.1 File Management Systems

File systems are the first generation database management system. At present, file systems are still commonly used in many engineering applications. The data model of a file system is called the file data model. Persistent data are described by declarations in the source language by listing the names of the fields of data in each record of a file. The declarations also describe the type and length of each field. Application programs use program language features (*READ*, *WRITE*, etc.) to interact with the operating system's file system to transfer data from the file to the work space and from the work space to the file, respectively. The access to records in a database file can be specified as either

sequential or direct to achieve access efficiency.

Application programmers are responsible for describing the format and sequence of data records in the database files. File systems meet only two of the requirements for a DBMS discussed in the preceding section: data persistence and secondary storage management. File systems are usually very efficient. Their possible problems have already been discussed in Section 13.1.

### 13.3.2 Hierarchical and Network Database Management Systems

Hierarchical and Network DBMSs are the second generation of DBMSs. They were developed in response to the basic need to have data storage that could be shared by multiple programs. The data models of these two types of DBMSs are the hierarchical data model and network data model respectively.

In the hierarchical data model, records of data are arranged in a hierarchy or tree structure according to parent-child relationships between records. A parent record may have many children records of various types, but a child record has exactly one parent record. The network data model removes this restriction in that records of data are arranged in a network of relationships and each record can have multiple parents.

These two types DBMSs commonly meet four of the basic requirements for a DBMS: data persistence, secondary storage management, concurrency control, and recovery. They also provide a limited data-independence in that programmers do not have to know the physical structures to access data records. Thus, they enable data sharing between different programs. These systems are characterized by: (1) having their own data definition and manipulation language; and (2) supporting multiple types of records interrelated in rigid structures. They can be quite efficient for the data access that they are designed to support.

These two types of DBMSs require that the application programs understand how the data records are logically organized in either a hierarchical tree or a network. Thus, if the logical data record organization is changed, the program referencing the data must also be changed. Moreover, the logical organizations of records in these systems are relatively hard to change and the development of new applications with an existing DBMS can be a time-consuming task.

### 13.3.3 Relational Database Management Systems

Relational database management systems (RDBMS) are the third generation of DBMSs. The important original objectives of the relational data model are simplicity, data independence, and rigor as described below.

- a. **Simplicity.** An RDBMS views data as if they were formatted into tables, which are called relations. A data record, also called a tuple, occupies a row in a table. The columns of the table represent the common properties of each record, also referred to as the table's fields or domains. A row in a table is identified by a primary key, which may comprise the values in one or more columns. A row's primary key value is by definition unique within the table. In many cases, it is quite natural to view information in terms of tables. Connections between tables are formed by columns with the same name of comparable values.
- b. **Data independence.** The programmer or end-user views and accesses the data as it is stored in tables while the underlying storage structure of a RDBMS may be some other data structure.
- c. **RDBMS is based on set theory and relational calculus, even though this theoretical foundation may not be important in the actual implementation of RDBMSs.**

Relational database management systems meet all these requirements for a DBMS. RDBMSs are very convenient for applications primarily to produce reports, such as marketing and banking applications. Typical relational database systems are DB2 developed by IBM, Ingres developed at University of California at Berkeley, and DBase developed by Microsoft. Relational DBMS have become very popular since the middle of 1980s (Loomis, 1990).

RDBMSs are characterized by a language interface called SQL which is used to define table structures and to access and update tables. SQL is the universal way to express retrievals, insertions, deletions, and updates in relational databases. SQL is a closed language in that it operates on tables and produces tables. SQL is typically supported in two modes: interactive and embedded in another language. The interactive mode stands alone in that no programming other than in SQL is required to access a relational database.

A typical RDBMS application is written using a combination of the SQL and a programming language such as C and FORTRAN. Any SQL statement that can be entered interactively can be alternatively embedded in a program. A special command translator called host language preprocessor is necessary. The preprocessor extracts the database manipulation commands written in SQL from the program and replaces them with calls to the run-time support unit of the database management system.

The design of a relational database is a process of determining first what tables are needed to represent application objects, and then optimizing those table structures by using the so-called "normalization" technique for efficient performance. This technique minimizes the duplications of data across tables and commonly results in several tables being required to represent a single application object. This normalization results in a weakness of RDBMSs for manipulation of complex objects. To manipulate complex objects, a RDBMS must join many tables. This is a time consuming RDBMS operation.

Another weakness of RDBMSs is the embedding of SQL in another language, and that SQL is table-oriented while other languages are typically record-oriented. This causes an impedance mismatch. This aspect will be discussed further in the next section.

#### 13.3.4 Problems with Older Generations of DBMSs

As discussed previously, flat-file management systems and hierarchical and network DBMSs meet partially the basic requirements, while the relational technology meets all the basic requirements for a DBMS. However, these older technologies have two common serious weaknesses: (1) a large semantic gap with applications dealing with complex data, and (2) an impedance mismatch between the data-manipulation language utilized by DBMSs and general-purpose programming languages.

##### 13.3.4.1 Semantic Gap

One of the measures of a DBMS's quality is how easily the system can be used to model real world entities into a collection of data to be handled by the system. The "distance" between the data models supported by the system and real world entities manipulated by applications software is referred to as the "semantic gap". The older DBMSs lack sophisticated data types. Only primary data types such as integer, real, and text string are supported. This is adequate for classical business applications in which the data used is for the most part very simple. Thus, the relational database systems are quite satisfactory for such applications.

However, many engineering applications deal with very complicated real world entities. The data structures representing these entities may contain fields



which are also data structures or references to other data structures. With the older technologies such as relational DBMS, these complicated entities have to be artificially fit into the relational tables. This leads not only to difficulties in the integration of an application with a DBMS, but also lower performance in data access.

#### 13.3.4.2 Impedance Mismatch

In developing database applications with older DBMSs, two languages are usually needed: a data-manipulation language, with the SQL of relational database systems as a typical example, and a general-purpose programming language in which a large portion of the application is written. This is because the SQL lacks the completeness to express the non-data-manipulation part of the application, and the general-purpose language has persistent data only in the form of files. Information must be passed between the two languages that are semantically and structurally different. This impedance mismatch is reflected in two aspects:

1. The difference in programming paradigms. SQL deals with a-table-at-a-time, while a common programming language usually deals with a-record-at-a-time. A loss of information may occur at the interface since the common programming language is often unable to represent relations (i.e., the database structures of SQL) directly.
2. The difference in type systems. Since there are two systems of data types, there is no automatic way to type check the application as a whole.

The impedance mismatch affects directly the software productivity. With a RDBMS, the typical software development process is divided into the following separated and disjoint activities: analysis and design, implementation in a certain programming language, and database definition and access in another

language SQL. Entities in the application domain must be translated first to the conceptual space of the implementation programming language and then translated to the conceptual space of the database management system.

With object-oriented DBMS, these two problems will be solved.

### 13.4 Object-Oriented Database Management Systems

#### 13.4.1 The Motivation

Object-oriented database management systems (ODBMSs) are the fourth generation of DBMSs. Object-oriented database management technology is the natural result of the union of two technologies: object-oriented programming and database management. The motivation for the development of ODBMS comes from the demand in support of data-intensive engineering applications such as Computer-Aided Design (CAD), Computer-Aided-Manufacturing (CAM), Computer-Aided-Engineering (CAE), and Computer-Aided-Software-Engineering (CASE).

These applications tend to center on high-performance graphic workstations and computation environments that support engineering activities. Such applications require massive amounts of persistent data. The level of complexity of these programs and of their data has grown far beyond what traditional database systems are prepared to handle. Commercial ODBMSs begin to appear in the market, and the current (1990) primary ODBMS vendors in the United States are Object Design, OBJECT-Science, Objectivity, Ontologic, and Servio Logic (Thomas, 1990).

The data model of ODBMS more closely matches real-world entities. The basic idea of ODBMS is to represent an entity in the real world being modeled with a corresponding item in the database. This modeling includes not only the data structures (referred to as properties) of objects, but also the operations

performed on the objects (referred to as the behavior of objects). Objects can be stored and manipulated directly, and there is no need to transform application objects into tables. Data types can be defined by users, rather than being constrained to some pre-defined types. This factor shortens greatly the semantic gap between the DBMS and applications having complex data, without sacrificing the performance efficiency. This is one of the major goals of ODBMS.

Another important goal of ODBMS is to provide database management support to object-oriented programming. An ODBMS is typically implemented in an object-oriented language and supports applications written in the same language (However, some ODBMSs support also other object-oriented languages and traditional procedural languages as well). Thus, the ODBMS can be integrated seamlessly with the object-oriented programming language and the programmer only deals with a single uniform model of objects. This results in a no-vault software engineering process and leads to a single, unified conceptual model used in all phases of development and maintenance. This avoids the impedance mismatch that frequently occurs in the older DBMSs and thus improves software productivity, quality, and flexibility.

ODBMSs are first and foremost database management systems. They have the capability to meet the basic requirements of modern database management systems. Other features distinguishing ODBMSs from older systems will be discussed in the next sub-section.

#### 13.4.2 Distinguishing Features

The distinguishing features of ODBMSs center on the idea of supporting object-oriented programming. They support the object-oriented programming notations of object classes, encapsulation, complex objects built by aggregations and inheritance, and polymorphism. ODBMSs enable objects to live outside the boundary of particular programs. Thus, ODBMSs are fundamental to the

viability of object-oriented development of many software systems and offers the potential to extend object-oriented programming to a broader scope than is impossible with object-oriented languages alone.

#### 13.4.2.1 Modeling Capability

An ODBMS allows any real-world entity to be uniformly modeled as an object, not as multiple tuples spread among several tables. An object belongs to a class or a data type. An ODBMS supports the definition of new object classes or new data types, rather than constraining programmers to use a fixed set of predefined types. New classes are indistinguishable from system-supplied classes for application programming: operations to objects of new classes are syntactically similar to and as efficient as the built-in operations on predefined classes.

An object class describes two aspects of objects: data structures and behavior. The data structure of an object contains the properties of the object. The value of an instance variable in a data structure can be of atomic types (integer, real, string etc.) and can be also an object. Thus, an ODBMS supports the definition of complex objects which are built up from aggregations of sub-objects. Behavior denotes the semantics of objects of the class. Operations performed on objects of the class may be defined to describe the semantics of objects, as well as extensive integrity constraints, domain constraints, and exceptions. Thus, with a DBMS, constraints on objects can be more easily and precisely specified. Also, the code which defines the operations on objects of the class and is previously in applications may be included in the database. This prompts reuse and sharing of the objects' semantics embedded in the code.

An ODBMS allows the data structures and behavior of objects to be encapsulated, since they can only be accessed or invoked from outside of the

object through message passing.

An ODBMS also allows classes to be organized in a class hierarchy by inheritance mechanisms to express multi-levels of abstraction. A class inherits all the properties and methods from its direct and indirect ancestors on the class hierarchy.

#### 13.4.2.2 Object Identifier

RDBMSs are value-based in that the identifier, called the primary key, of a row in a RDBMS is based on value of field(s) in the row. When the value of a primary key field changes, the row becomes a different row. If a row is used to model an object, changing the value of a primary key field would not keep the row affiliated with that object.

ODBMSs are identifier-based in that the object identifiers are independent of the particular values of instance variables in the objects. Also, an object's identifier is different from and independent of the programming language symbol used to refer to the object. The identifier is the property of an object that distinguishes it from other objects and remains invariant across all possible modifications of its instance variables.

#### 13.4.2.3 Transparent Database Transaction

Some ODBMSs are tightly integrated with object-oriented languages and have the capability for both primary and secondary memory management. Objects can be stored and retrieved automatically by such systems. The location of objects and the movement of objects between primary memory and secondary memory are transparent to the application programmer. Thus, with

such systems, there is no boundary between objects in primary memory and those in persistent store. Persistent objects and transient objects can be treated equally. Such systems offer a further reduction in application development efforts.

### 13.4.3 Implementation Approaches

At the present, there are three major approaches to the development of object-oriented database management systems: RDBMSs with object interfaces, linkable ODBMSs, and systems which add persistence to object-oriented languages (Thomas, 1990).

#### 13.4.3.1 RDBMSs with Object Interfaces

These systems are extensions of relational database systems to support object management. They provide object-oriented interfaces such that applications can communicate with them in terms of objects. These systems are used as a storage services for object-oriented systems and, internally, they do not have object-oriented features. Objects in such systems are still stored only as data without the associated methods and the objects are actually mapped to and from relational data types. This approach is a step forward but does not solve many of the problems of RDBMSs since data modeling capability is still limited and somewhat unnatural (Thomas, 1990).

Postgres, a research project at the University of California at Berkeley, is a typical example of this type. Postgres is developed based on the RDBMS Ingres, and it extends the relational data model with an abstract data typing mechanism and identifies procedures as a fundamental data type. Support for abstract data types means the users are allowed to define a new data type, use

that type to define a column in a relation, and define procedures which are usable in relational queries that manipulate the new data type. It makes some provisions for objects, but does so through storing its SQL and C procedures as attribute values in tables.

#### 13.4.3.2 Application Linkable ODBMSs

Application linkable DBMSs are at the mainstream at present in the development of ODBMSs. These systems provide shared storage for objects and may be used with either object-oriented or traditional procedural languages. Applications are built on top of the ODBMSs, and they use the facilities provided by the ODBMSs through interface functions and other ODBMS specific mechanisms.

Gemstone by Servio Logic (Maier et al., 1986, 1987) is the first ODBMS of this sort. Gemstone is a Smalltalk based system and provides an extension of Smalltalk, called OPAL and used for data definition, data manipulation and general computation. Gemstone consists of an object-server and a workstation interface. The interface supports either C or Smalltalk applications running on Sun or PC-based workstations.

ONTOS (Ontologic Inc., 1990) provides a complete C++ based development environment for object-oriented applications. At the center of the environment is the ONTOS Object Database supporting the storage of many different types of data. ONTOS provides a transparent interface to C++ applications. A class named *Object* is defined in the ONTOS class library and used as the base for all persistent classes. The *Object* class defines a set of methods to create and destroy persistent objects. Each persistent object created has a unique identifier within an ONTOS database. During the execution of an application, persistent objects are automatically loaded from databases to

virtual memory when the objects are needed. ONTOS also provides a set of tools including schema design tools and graphical user interface tools for the development of applications. The schema loader of the ONTOS' environment operates on an application's C++ class definitions directly and generates schemas for persistent classes. Thus, no data definition and data manipulation language (DDL and DML) preprocessors are needed.

ObjectStore (Object Design Inc., 1990) has just appeared on commercial market in 1990. ObjectStore is based on the AT&T's C++ specification and has three principal components: the runtime unit, C and C++ application library interfaces, and C++ development tools. ObjectStore provides a complete distributed DBMS services by its runtime unit. It adds several additional instructions to the syntax of C++ including the *persistent* instruction to declare persistent storage class, and the *transaction* instruction to retrieve objects from the persistent store. These additional instructions will be processed by a DML preprocessor compatible with Cfront of the AT&T C++ compiler. The DML preprocessor also supports parameterized types that strengthen the AT&T C++. ObjectStore provides an OQL (object-oriented query language) for applications to retrieve objects in an object-oriented fashion. It uses a Virtual Memory Mapping Architecture and allows persistent objects to be handled in the same way as transient objects in programming and, therefore, provides a certain degree of transaction transparency. The developers of ObjectStore claim that ObjectStore has the capability to reduce application development efforts to a great degree.

#### 13.4.3.3 Adding Object Persistency to Object-Oriented Languages

Adding object persistency to object-oriented languages seems to be the best approach for maintaining long lived objects. Such languages should be computationally complete in that all the computation required in an application



can be written in the language, and persistence should be simply another characteristic of an object. Transparent database transaction should be supported. Thus, such languages will provide a truly seamless approach in the integration of the object-oriented paradigm with database management and avoid impedance mismatches since there is a single object model for everything in the system. This is still an area of research in object-oriented language. ObServer, Persistent Smalltalk, Mnome, and Sticky are examples of current research (Thomas, 1990).

It is worth mentioning that in the language Eiffel (Meyer, 1988), object persistency is offered by using a library class *STORABLE* with methods *store* and *retrieve*. If a class is a descendant of *STORABLE*, and if *fn* is a file name, an object of the class can be stored by the instruction *x.store(fn)*, and retrieved by *x.retrieve(fn)*. The entire object structure referred to by *x*, directly or indirectly, will be stored or retrieved. The external representation preserves the references. By choosing the right *x*, an entire object structure or part of it can be stored or retrieved. Persistent objects can be shared, however, the classes which store and retrieve the objects have to be the same.

#### 13.4.4 Limitations of Current ODBMSs

At present, object-oriented database management technology is still in its infancy and continuously evolving. It is hard to evaluate the current commercially available systems without studying the complete documentation which can only be obtained with the purchase of these systems. With limited literatures available, several limitations of these systems are summarized in this section.

#### 13.4.4.1 Multiple Viewing of Persistent Objects

Data-independence is one of the primary goals of database management systems. ODBMSs are attaining the goal in terms of objects: persistent objects of a certain class can be shared by applications declaring the same object class. In engineering data management, all the information about an entity should be ideally defined and stored as one object for the sake of consistency. However, each of the applications sharing this information may see this entity from a different perspective and access only to a portion of this information. Thus, for the same set of information, there may be many different object representations: one for the complete information stored in a database, and others for different perspectives of each individual application. In other words, a persistent object may have multiple views, and each view results in a particular object definition.

An ideal ODBMS for engineering software systems should have the capability to handle the view transformation between databases and applications. The following types of view transformations should be supported:

1. A view can be defined according to the inheritance hierarchy of object classes: an object of class *B* should be able to be retrieved as an object of class *A*, where class *A* is a base class of class *B*.
2. A view can be defined by hiding named fields of an objects and these hidden fields may be distributed over levels of inheritance hierarchy.
3. A view can contain new fields derived (computed) from the hidden fields of an object.

This capability, however, has not been well addressed and supported by the ODBMSs currently available on commercial market. The storage of objects on disk depends upon a fixed representation of the objects. Thus, if the class definition of objects changes (i.e., if instance variables are added or deleted), the systems are not able to load the old objects without using programs to convert

the objects. For the same reason, multiple views on the same objects is not supported by these systems.

#### 13.4.4.2 Provision of In-Core Object Management

To be able to communicate to each other, code units in an application, object classes or object sub-systems, have to know the objects they need to communicate with. As discussed previously, to prompt software flexibility, extensibility and standardization, the use of objects with global scope is generally prohibited. A code unit should be able to inquire an ODBMS to obtain the object defined elsewhere to which the code unit need to communicate with. The code unit should be able to do so as long as the object exists regardless whether this object is in the primary memory (in-core) or in the secondary memory.

Moreover, some ODBMSs support transparent database transaction that objects can be stored and retrieved automatically. This is convenient for many applications because programmers are relieved from performing object transaction explicitly. However, for applications which deal with large objects or large number of objects, the programmers should be able to control the object transaction to improve the performance of such applications. This is because the programmers know better than the ODBMS which objects should be kept in the primary memory and which objects should be purged from the primary memory during the execution of applications. Thus, facilities should be provided by an ideal ODBMS such that a mixed control on object transaction may be accomplished. The programmers may take the advantages of the transparent object transaction provided by the ODBMS for some objects and may also perform the transaction explicitly for other objects in applications.

The facilities mentioned above, however, have not been well addressed and supported by the ODBMSs currently on the commercial market, as seen from the available literature of these systems.

### 13.5 Engineering Data

Engineering is a multi-disciplinary activity. For example, in building construction, several activities are involved including finance planning, conceptual planning, design, analysis, and construction management. These activities cover accounting, architecture, structural, mechanical, electrical, and foundation engineering, and construction engineering. To automate these activities utilizing the computing power provided by modern computer technology, an integrated system consisting of a number of applications is necessary. These applications share a common set of information stored in a central database, as illustrated in Fig. 13.2, and generate a huge amount of new information required to store in databases. Engineering data here refers to these information.

Differing from business data, engineering data has the following features:

1. **Complexity:** engineering data often can not be represented in tables due to the complicated inter-relationships between different kinds of data.
2. **Different views from the perspective of different disciplines:** different applications in an integrated system may require substantially different views of the shared data. For example, in designing a building column, architectural engineers are interested in the geometry and color of the column, structural engineers are concerned about the geometry, section modulus, stress and deformation of the column, and designers are concerned with the geometry and the type of the standard rolled shape of the column.

3. **Volatility:** different engineering data may have different lifetime. For example, the data used for and generated from structural analysis software can be seen as derived data from the primary data. These data are highly volatile since only a small fraction of the data need be stored in a permanent database after the analysis. However, this volatile data should be stored during the analysis activity. The data describing structural geometry and design specifications are of a more permanent nature.
4. **Constraints:** engineering data is often subjected to many constraints. Some constraints may not be simply expressed as allowable ranges of a certain value. Application code is often needed to perform the validation checking.

An ideal DBMS for engineering data should have the capability of handling these features in a simple and natural way.

### 13.6 Overview of DBMSs for Engineering Software

The importance of DBMSs for engineering computation has been recognized for many years, and many approaches have been proposed or developed.

Felippa (1989) has developed several database systems for primary and secondary memory management. These systems are mainly for supporting finite element analysis in a FORTRAN programming environment and of hierarchical and network database management types.

A system called EDIPAS (Engineering Data Interactive Presentation and Analysis System) has been developed at the National Aerospace Laboratory of the Netherlands (Steenbergen et al., 1986). It is built around a commercially available DBMS. It is a sophisticated hierarchical database management system. The basic entity of EDIPAS is the datablock. One datablock contains

interrelated data defined by the user. Each data item, a scalar or a matrix, is identified by a name. Data blocks are organized in one or more multi-level hierarchical structures.

An extended relational model is used in the work by Arora et al. (Arora et al., 1988; Mukhopadhyay et al., 1987; Murthy et al., 1986a, 1986b) at the University of Iowa. A relation is seen as a two-dimensional array and each of the columns of the array has a unique definition. Thus the concept of a relational table is generalized such that a matrix can be seen as a special case of a relational table.

These existing systems are of older generation systems and can not efficiently handle complicated engineering data. An object-oriented database management system is the best choice at the present time.

Powell et al. (1988a, 1988b) have envisioned an integrated software system for structural design activities based on object-oriented programming and data management concepts. The architecture of such a system is illustrated in Figure 13.3. This system has an object-oriented central database containing a repository for a geometric model of the structure, material property data, and other data that must be shared by the application programs. This database ensures overall consistency of data, but does not have application-specific capabilities. The database is under the control of a full-featured DBMS.

To perform design operations, objects are extracted from the central database, and operated on by an application program in a local or application-specific database. The local database also stores large amounts of data specific to the application. When the activities of the application are successfully performed, desired objects in the local database are reinserted to the central database. Thus, objects are moved between the central database, the local database and the applications. The central database is concerned with maintaining a common view of each object, from which applications can extract

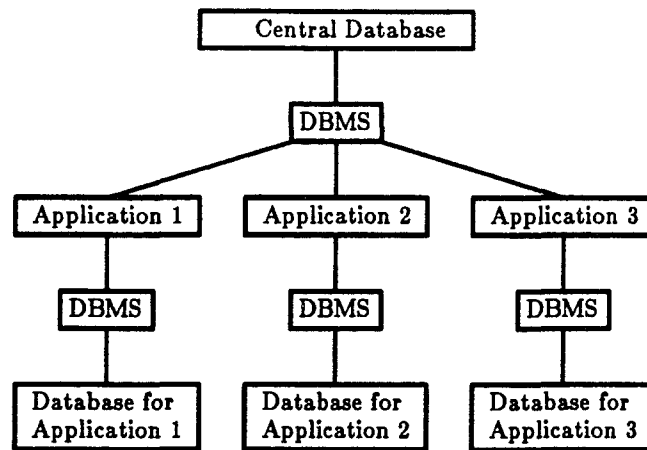


Figure 13.3 Configuration of an integrated system with local databases

specific views. It maintains the overall consistency of the data. The system envisioned by Powell provides a viable approach for integrated engineering software systems.

### 13.7 Integrating an ODBMS with the SESDE

The development of an ODBMS for the use in the SESDE is a major effort and cannot be done in a limited time in a university environment. It is therefore not feasible nor necessary to develop the DBMS component here for the SESDE from scratch. A commercial ODBMS such as the ObjectStore by Object Design or Ontos by Ontologic should be utilized to establish the data management facilities in the SESDE. Along this line, some necessary facilities need be developed around the commercial ODBMS for the integration.

### 13.7.1 A View Transformation Manager

One of the facilities is the object View Transformation Manager (VTM). The VTM should be built on the top of the ODBMS and be a general facility capable of transforming objects of any class from one view to another. An object description language must be defined so that the characteristics such as type, length, range, etc. of named fields (or instance variables) of object classes can be described. A main view can be defined in the language for a class of persistent objects stored in a database containing the complete information of all fields in objects of the class. Other views of the same set of objects can also be defined in the language describing the relationships between the named fields in a particular view with the fields in the main view. The VTM interprets the descriptions written in the language, retrieves objects from the database, and generates the objects of the desired class. These generated objects will be stored in another database or be passed to a particular application.

With the VTM, the configuration of a typical integrated system is shown in Fig. 13.4. The integrated system will have a central database containing the completed and shared information of the objects shared by the applications of the system. Each application may have its own local database storing objects generated from views of the objects stored in the central database and objects specifically for the particular application. The VTM transforms objects between the local and the central databases according to the description of these object classes. All central and local databases are managed by the ODBMS.

### 13.7.2 An Input Manager

The other facility is an Input Manager. In most recent applications, computer graphics and graphical user interfaces are utilized to input data graphically. However, in many cases it is still necessary for applications to input data textually, i.e., input from ASCII files. An application usually requires the



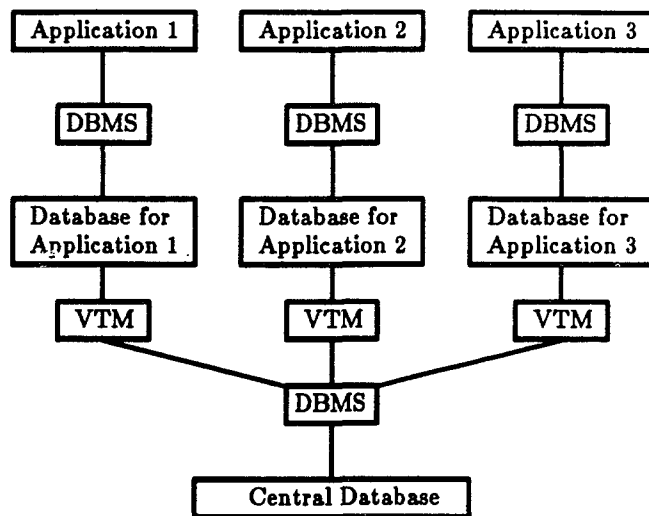


Figure 13.4 Configuration of an integrated system with local databases and a view transformation manager

input data in a file ordered in a strict predefined sequence so that the application can read the data accordingly. Thus, the portion of code dealing with data input and the data file are tightly bound with this predefined sequence. Moreover, this portion of code also have to perform validation checking of input data. It is the most cumbersome and error-prone portion in many programs.

To release the strict correspondence between the code dealing with data input and the input data sequence, an input manager is necessary. The input manager is responsible for handling the data input and validation process for applications. It interprets the definition of object classes written in the object description language discussed in the preceding sub-section, reads objects of these classes from data files according to the definitions, and generates objects of the desired classes. These generated objects may be stored in a database via the ODBMS. It should also handle data of atomic types as well. A particular application may inquire the ODBMS to obtain the necessary data objects without explicitly processing the data files.

### 13.7.3 In-Core Object Management

As discussed previously, in-core object management includes transferring objects between code units and mixed control of object transaction. This capability can be implemented by either enhancing the ODBMS integrated with the SESDE or building facilities based on the ODBMS. This depends on the specific features of the particular ODBMS integrated with the SESDE.

## LIST OF REFERENCES

1. Arora, J.S., and Mukhopadhyay, S., "An integrated Data Base System for Engineering Applications Based on an Extended Relational Model", *Engineering with Computers*, No. 4, 1988, pp.65-73.
2. Colton, J.S., "The Design and Implementation of a Relational Materials Property Data Base", *Engineering with Computers*, No. 4, 1988, pp.87-97.
3. Dawson, J., "A Family of Models", *BYTE*, September 1989, pp.277-286.
4. Felippa, C. A., "Implementation of Scientific Data Management in Computational Mechanics: Personal Experiences", in *State-of-the-Art Surveys on Computational Mechanics*, Edited by A.K. Noor and J.T. Orden, ASME, 1989, pp.469-491.
5. Kim, W., "Object-Oriented Approach to Managing Statical and Scientifical Databases", in *Statical and Scientific Database Management, Proceedings of Fifth International Conference, V SSDBM, Charlotte, N.C., USA, April 1990*, pp.1-13, Edited by Michalewica, Z., Springer-Verlag, New York, 1990.
6. Loomis, M.E.S., "The Basics", *Journal of Object-Oriented Programming*, May/June, 1990, pp.77-81.
7. Loomis, M.E.S., "ODBMS vs. Relational", *Journal of Object-Oriented Programming*, July/August, 1990, pp.79-82.
8. Maier, D., Stein, J., Otis, A., and Purdy, A., "Development of an Object-Oriented DBMS", *Proc. OOPSLA, 1986*, pp.472-482.
9. Maier, D., and Stein, J., "Development and Implementation of an Object-Oriented DBMS", in *Research Directions in Object-Oriented Programming*, MIT Press, 1987, pp.355-392.
10. Meyer, B., "Object-Oriented Software Construction", Prentice Hall, New York, 1988, 534pp.
11. Moss, J.E.B., and Sinofsky, S., "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface", *Advances in Object-Oriented Database Systems, Proceedings of 2nd International Workshop on Object-Oriented Database Systems*, Springer-Verlag, 1988, New York, pp.298-316.

12. Mukhopadhyay, S., and Arora, J.S., "Design and Implementation Issues in an Integrated Database Management System for Engineering Design Environment", *Advances of Engineering Software*, Vol. 9, No. 4, 1987, pp.186-193.
13. Murthy, T.S., Shyy, Y.-K., and Arora, J.S., "MIDAS: Management of information for design and analysis programs", *Advances of Engineering Software*, Vol. 8, No. 3, 1986a, pp.149-158.
14. Murthy, T.S., and Arora, J.S., "Database Management Concepts in Computer-Aided design Optimization", *Advances of Engineering Software*, Vol. 8, No. 2, 1986b, pp.88-97.
15. Object Design Inc., "ObjectStore Technical Overview", Version 1.0, 1990.
16. Ontologic Inc., "The ONTOS Development Environment", 1990.
17. Peterson, R.W., "Object-Oriented Data Base Design", *AI Expert*, March 1987, pp.180-188.
18. Powell, G.H., Bhateja, R., Abdalla, G., An-Nashif, H., Martin, K., and Sause, R., "A Database Concept for Computer Integrated Structural Engineering Design", in *Computing in Civil Engineering: Microcomputers to Supercomputers*, Proceedings of the Fifth Conference, ASCE, Edited by Kenneth M.W., 1988a, pp.521-529.
19. Powell, G.H., and Bhateja, R., "Data Base Design for Computer-Integrated Structural Engineering", *Engineering with Computers*, No. 4, 1988b, pp.135-143.
20. Steenbergen, H., and Heerema, F.J., "Database Administration Facilities for Engineering Data Management", *Advances of Engineering Software*, Vol. 8, No. 3, 1986, pp.141-148.
21. Thomas, D., "Object-Oriented Databases and Persistent Objects", *Journal of Object-Oriented Programming*, July/August, 1990, pp.59-60.
22. Zdonik, S.B., and Maier, D., "Fundamentals of Object-Oriented Databases", in *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990, pp.1-32.

## CHAPTER 14 SUMMARY, CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK ON SESDE

### 14.1 The Problems and the Solution

The critical issue addressed in this work is the reusability of software for research and instruction in a university environment. Because of the lack of reusability of existing software, the development of research and instructional software is usually slow, time consuming, and low in quality. This work is directed at the design and development of the framework for a domain-specific Structural Engineering Software Development Environment (SESDE) as a solution to the software problems in structural engineering computing. The completion of SESDE should provide a systematic support for software reuse and also serve as a crucial layer between structural engineering applications and the evolving computer technology.

The software development approach adopted in this work is quite different from the traditional structured programming approach. Most existing research and instructional software systems in structural engineering were developed using the traditional approach. As a result, these programs are often one-of-a-kind in that the components of a program are specially designed for a specific application. These components are not generally reusable or may be reused only at a very low level. In most cases, each application is developed from scratch and can not take full advantage of previous development. This leads to the following common problems: long development times, low quality, and difficult maintenance. Also, these programs may be hard to survive or require a considerable amount of time to modify, if the computing environment (hardware, operating system, etc.) on which these systems are based is changed.

Herein, the focus is on object-oriented programming and software reusability. Object-oriented programming methodology emphasizes the abstractions of entities in a specific domain rather than each individual application. Each abstraction may be implemented as an object class or a reusable component, and the relations between these abstractions can be utilized in the design and implementation of reusable components. Software development efforts are intentionally accumulated in terms of reusable components and are capable of being utilized efficiently in further development. A specific application in the domain can be constructed utilizing these reusable components, and new reusable components may be developed along with the implementation of the specific application. This approach leads to higher software reusability, higher productivity, quality, flexibility for modification and extensibility. However, this approach requires a large start-up effort to create a set of basic reusable components for a specific domain.

Based on the concept of software reusability and to provide a systematic support to software development in the domain of structural engineering, a Structural Engineering Software Development Environment (SESDE) has been envisioned in the present work. The architecture of SESDE is shown in Figure 14.1 where the components of the SESDE are enclosed in the dashed box. SESDE consists of reusable components (solid boxes in Fig. 14.1) and CASE tools (dotted boxes). The reusable components are classified in the following four groups:

1. A Graphical User Interface Development System (GUIDES). GUIDES implements a set of GUI tools for applications to build their interface.
2. An Object-oriented Database Management System (ODBMS). ODBMS provides the database management facility for applications.
3. A Generic Object Class Library. This library contains general-purpose object classes whose use is not limited to structural engineering.

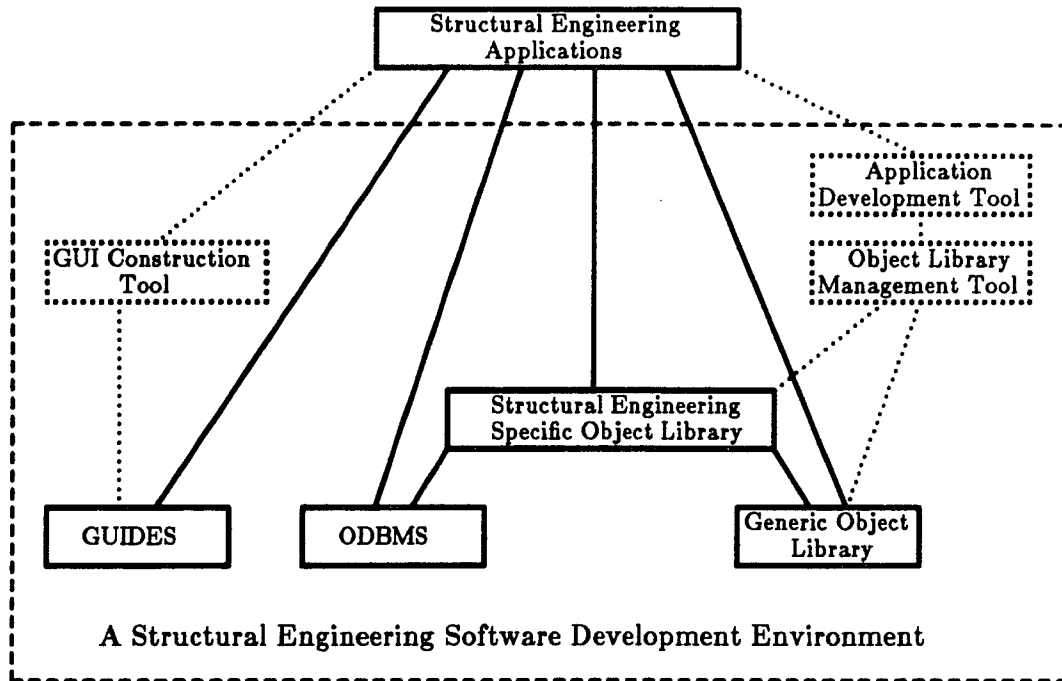


Figure 14.1 Architecture of the SESDE

4. A Structural Engineering Specific Object Class Library. Specific sets of object classes are contained in this library to facilitate structural engineering applications software development.

Three CASE tools are envisioned for the SESDE: (1) A GUI construction tool, (2) An object class library management tool, and (3) A structural engineering application development tool.

Present achievements on development of the SESDE are summarized in the next section.

## 14.2 Summary of the Present Work on SESDE

The complete development of the SESDE requires a major effort. The present work has built a basic framework of the SESDE and established a model of such an environment for other engineering areas. The architecture and major components of this environment have been identified as shown in Figure 14.1. General requirements for various components have been established. GUIDES and a set of classes in the generic object class library have been fully designed and implemented. The present work provides a foundation for a complete development of the SESDE. The tasks achieved are listed in the following subsections.

### 14.2.1 The Graphical User Interface Development System

Interactive graphical user interfaces are an essential part of modern engineering software. However, the code which handles the graphical user-interface is often complex and difficult to debug and modify. It accounts for a significant portion of the code of interactive graphics applications. Therefore, the design and implementation of the user interface of a program is a very important but difficult task. A Graphical User Interface Development System, GUIDES, which is a set of reusable components, has been developed in the present work to allow rapid generation and modification of graphical user interfaces, and to provide a crucial layer between applications software and the various evolving user-interface environments.

The development of GUIDES has been described in PART TWO. GUIDES is similar in many aspects with the emerging industry GUI standard OSF/Motif. GUIDES provides programmers with a reasonably complete set of user-interface tools such as menus and dialogue boxes. It provides an Interface Description Language to achieve a better separation between the user-interface and other components of an application. Applications can use this language to



specify their user-interfaces independently of the application-specific code. Applications communicate with GUIDES at runtime through a procedural interface such that GUIDES is usable with both procedural languages (such as C and FORTRAN) and object-oriented languages (such as C++).

GUIDES have two important features which have not been well addressed in Motif: (1) GUIDES works with a three-dimensional interactive graphics package naturally and compatibly; and (2) Not only the static appearance but also the dynamic behavior of a GUI may be specified in the GUIDES description language. It is expected that any application based on GUIDES can be easily modified in the future to interface with a standard GUI system that combines with a three-dimensional interactive graphics capability.

In current structural engineering applications, GUIDES has been used in the development of STARPAC: a finite element analysis response visualization program. Furthermore, GUIDES has been used presently in the Computer Graphics course taught at the Civil Engineering School of Purdue University as an illustration of GUI technology.

#### 14.2.2 The Generic Object Class Library

A set of classes in the generic object class library has been developed in the C++ language and described in PART THREE. These classes represent and implement commonly used entities and utilities in engineering software. This development demonstrates the feasibility of object-oriented approaches for engineering software, and also provides models for the development of other reusable components. These classes can be subdivided into three groups:

1. Object classes for general data structures and general utilities. These include classes for general data structures such as text string, vector, extensible array, and for general utilities such as error-handling and on-

line argument processing utilities.

2. Object classes for full matrices. Six classes are included in this group. A *Matrix* class is developed which represents general full matrices and implements general matrix operations. Five other classes are developed as derived classes of the *Matrix* class. Each of the derived classes represents matrices with a specific characteristic and implements related operations. Operators are overloaded with most matrix operations such that these operations may be coded in a more expressive and abstract fashion.
3. Object classes for sparse matrices. Two abstract classes are developed to represent two types of sparse matrix abstractions: unknown-based and node-based. Derived from the two abstract classes respectively, several classes are developed which implement different sparse matrix storage schemes including the active-column scheme and the graph-based scheme.

Object classes in this library are basic reusable components. They may be used readily in any research and instructional software development based upon the C++ language. They may be used directly in application programs, or be used to build more complex generic and structural engineering specific object classes. For example, the full matrix classes may be used directly to develop programs where matrix operation is the major operation, and may be used in building an element class for finite element analysis.

### 14.2.3 The Object-oriented Database Management System

A database management system in an integrated engineering computing environment is responsible for the data transfer from the user to an application, between code units in an application program, and between different applications. This system is vital to the development of standardized reusable components and to the integration of reusable components into applications.

The characteristics of engineering data and the database management requirements for integrated engineering computation environments have been critically reviewed and discussed in Chapter 13. It is concluded that the object-oriented database management should be used for integrated engineering computation environments. However, the database management aspects of the SESDE have not been implemented in the current research. The development of an ODBMS for the use in the SESDE is a major effort. A commercial ODBMS should probably be integrated and adapted to support the features of the SESDE. However, since such an ODBMS is not readily available at Purdue at the present time, no development work regarding the integration of an ODBMS with the SESDE has been attempted here, but specific issues associated with the integration have been given.

### 14.3 Specific Recommendations for Follow-up Work on SESDE

The SESDE has already made some impacts on research and instructional software development in structural engineering computing at Purdue, but its full potential can be realized only when a large number of reusable components are developed in the follow-up work. The necessary follow-up work of the SESDE should focus on both long-term development and short-term application of the SESDE components. To this end, the specific tasks in long-term and in short term are suggested in the following sub-sections.

#### 14.3.1 Long-term Tasks

The goal of long-term tasks is to complete the SESDE system development. Regarding to each major group of reusable components and CASE tools, the following tasks are suggested.

1. The Graphical User Interface Development System: GUIDES can be further enhanced by: (1) the development of more user-interface tools utilizing existing GUIDES agents; (2) the modification of the GUIDES code to utilize the CLOOP utility in order to improve the extensibility of GUIDES agents.
  
2. The Object-oriented Database Management System: The ODBMS is essential for the development of structural engineering specific components. Structural engineering specific components, including such components used in finite element analysis, may be developed without an ODBMS. However, the ODBMS is a must for the standardization, flexibility and extensibility of these components. The development of an ODBMS for the SESDE can better be approached by integrating a commercial ODBMS. However, if this integration is not possible, an ODBMS with limited basic functionality may be developed. As a direct application of the ODBMS and GUIDES, a management program of a semi-rigid beam-to-column steel building connections databank may be developed since a substantial amount of data for semi-rigid beam-to-column steel building connections has already been collected at Purdue University.
  
3. The Structural Engineering Specific Object Class Library: Many components in this library may be developed with a project of a finite element platform. This platform is for finite element testing, development, and analysis. This platform composed of reusable software components or object classes for finite element analysis may include the following sets of classes for: (1) structural and solid elements; (2) constitutive models; (3) linear, nonlinear, and transient analysis methods; (4) global solution algorithms; and (5) schemes for integration of rate-constitutive equations for finite element analysis. These classes can be selected and combined to build various executable programs involving any specific combination of

finite element techniques.

4. **The Generic Object Class Library:** A set of generic object classes has been developed in the present work. These classes may be used as basic building blocks in the development of structural engineering specific object classes as mentioned above. In the future development of specific classes and specific applications, effort need to be spent to explore the abstractions with general characteristics. Such abstractions may be developed as generic object classes and stored in this library.
5. **The CASE Tools:** The development of CASE tools of the SESDE presently has a lower priority than of the reusable components. However, when a large number of reusable components are developed, the development of the object class library management tool will become essential. Also, in order to prompt the use of GUIDES, the graphical user interface construction tool will be necessary to release application developers from learning the GUIDES description language.

Object-oriented programming and software reusability technique have shown a great potential in engineering software development, maintenance, modification and extension. However, to realize this potential, engineering software developers have to be well trained in software engineering, object-oriented programming and object-oriented languages. This is essential for the long-term development of the SESDE.

#### 14.3.2 Short-term Tasks

The short-term tasks are focussed on the promotion of the use of existing reusable components. At the present time, the foreseeable short-term tasks are around the application of GUIDES and its associated utilities. These tasks are described in the following:

1. The development of the FORTRAN Language interface for GUIDES. An attempt has been made in the design of GUIDES to make it usable with both C and FORTRAN languages. However, a complete FORTRAN interface of GUIDES has not been made available in the present work. Since FORTRAN is still the major language for engineering applications, the FORTRAN interface is an urgent need at the present.
2. The development of graphical utilities associated with GUIDES. Graphical utilities are graphical tools implementing specific functionalities commonly used in engineering applications. Typical examples are an X-Y Plot Manager and a File Manager. They should have procedural interfaces, the same as the GUIDES, and may be used as "black-boxes" in procedural languages. Used together with GUIDES, these utilities may facilitate greatly application software development.
3. The development of pre- and post-processing programs utilizing GUIDES and GUIDES associated graphical utilities for structural analysis programs. This will serve to further test the concepts and approaches that have been designed and implemented in the present work. Several programs of this type may be developed for different types of structural analysis such as two-dimensional finite element analysis and frame analysis. A pre-processing program allows the end users to define and to modify the definition of a structural analysis problem interactively and graphically. A post-processing program allows the end users to display the analysis results interactively. These programs may communicate with analysis programs through flat-files with pre-determined formats for the time being before the ODBMS becomes available.

VITA

## VITA

Hong Zhang was born in Beijing, China, on May 17, 1952. He received both his Bachelors degree in Solid Mechanics in 1977 and his Master of Science degree in Structural Geology and Geomechanics in 1982 from Peking University. After working at Peking University for four years, he continued his graduate study at Purdue University in August, 1986, and received his Ph.D. degree in May, 1991.