## 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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Order Number 9126854

# Object oriented paradigms for computer aided structural engineering

Yoon, Chong-Yul, Ph.D.

University of California, Berkeley, 1990

Copyright ©1990 by Yoon, Chong-Yul. All rights reserved.

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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Object Oriented Paradigms for**
**Computer Aided Structural Engineering**

**By**

**Chong-Yul Yoon**

**B.S. (Massachusetts Institute of Technology) 1980**
**M.S. (Massachusetts Institute of Technology) 1982**

**DISSERTATION**

**Submitted in partial satisfaction of the requirements for the degree of**

**DOCTOR OF PHILOSOPHY**

**in**

**ENGINEERING**
**CIVIL ENGINEERING**

**in the**

**GRADUATE DIVISION**

**of the**

**UNIVERSITY OF CALIFORNIA at BERKELEY**

**Approved:**
**Chair:** .................................... 8/27/90 ......
**Date**
.................................... 8/27/90 ......
.................................... 11/19/90 ......

**DOCTORAL DEGREE CONFERRED**
**DECEMBER 19, 1990**

**********************************

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Object Oriented Paradigms**

**for Computer Aided Structural Engineering**

**By**

**Chong-Yul Yoon**

**Abstract**

The increase in use of computers in many aspects of engineering constructed facilities has been enormous during the past few decades. Computers are used for analyses of structures, design checks and optimization, cost estimation, construction planning, etc. Using proven algorithms, many of the individual tasks are now highly automated. Engineering, however, involves interaction among many of these tasks and productive structural engineering systems must be an integrated software system. This dissertation deals with the fundamental interface to the resources in a computer system that engineers need to face in developing an integrated engineering system; specifically, the interfaces are a programming paradigm and a data model to represent engineering information in the database.

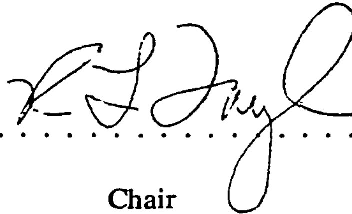
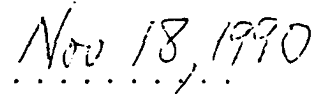
Object oriented concepts and paradigms have recently emerged as a promising theme in developing large systems. The objectives of the present study are to develop an object oriented software design method and an object oriented data model that are to become fundamental tools in developing large and integrated engineering systems.

The dissertation begins by introducing object oriented concepts. It then evaluates languages for engineering software development. The commonly used procedural languages such as Fortran and C are discussed. Various object oriented languages are considered and the C++ language is proposed as an appropriate language to develop today's engineering application programs. This is followed by the design and implementation of an object oriented finite element program written in C++. Based on experience in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementing this finite element program, a general guideline for object oriented development of engineering software using C++ is given. This guideline emphasizes levels of abstraction and reusability. A programming paradigm is an essential interface to the resources in a computer; the other interface is a data model that can effectively represent engineering information in the central database of the integrated system. A simple object oriented data model appropriate for engineering information is proposed in this dissertation. The model treats classes as objects and message sending is the only mechanism necessary for communication among data objects, database users and database administrators. This makes the model simple and uniform. The dissertation concludes with a summary of the work and recommendations for future research.

Approved:  

Chair Date

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Object Oriented Paradigms for**

**Computer Aided Structural Engineering**

**Copyright c 1990**

**Chong-Yul Yoon**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# DEDICATION

**To my parents**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Professor Robert L. Taylor for his guidance, motivation and continuous support during the course of this research. I would also like to thank Professors Gregory L. Fenves and Alice M. Agogino for serving on the dissertation committee.

I would like to express my deep gratitude to Professor Anil K. Chopra who guided my graduate studies at Berkeley and who was a constant motivating force for me.

My thanks to fellow students in Berkeley for their support, friendship and encouragement during my studies at the University of California at Berkeley. A special word of gratitude to Malcolm Gaustad for his editorial advice.

I am forever indebted to my parents, whose continuous love and support made this dream a reality. This dissertation is dedicated to them.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Table of Contents

Abstract

Acknowledgements ........ iv

Table of Contents ........ v

CHAPTER 1. INTRODUCTION

1.1. Computer Integrated Structural Engineering ........ 1

1.2. Objectives and Scope ........ 4

1.3. Dissertation Organization ........ 5

CHAPTER 2. OBJECT ORIENTED CONCEPTS

2.1. Introduction ........ 7

2.2. Basic Object Oriented Concepts ........ 8

2.2.1. Objects, Classes, Methods, and Messages ........ 8

2.2.2. Encapsulation ........ 9

2.3. Object Oriented Techniques ........ 10

2.3.1. Inheritance ........ 10

2.3.1.1. Derived and Base Class Relationships ........ 10

2.3.1.2. Single, Partial and Multiple Inheritance ........ 11

2.3.2. Polymorphism ........ 12

2.3.2.1. Polymorphism via Inheritance ........ 12

2.3.2.2. Overloading ........ 12

2.3.2.3. Late Binding ........ 13

CHAPTER 3. PROGRAMMING LANGUAGES FOR ENGINEERING SOFTWARE

3.1. Introduction ........ 15

3.2. Programming Principles ........ 16

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.1. Programming Styles .......... 16

3.2.2. Programming Paradigms .......... 19

3.2.2.1. Procedural Paradigm .......... 20

3.2.2.2. Data Hiding Paradigm .......... 22

3.2.2.3. Data Abstraction Paradigm .......... 23

3.2.2.4. Object Oriented Paradigm .......... 25

3.2.2.5. Other Approaches to Programming .......... 32

3.3. Programming Languages for Engineering software .......... 34

3.3.1. Deficiencies in Fortran .......... 36

3.3.2. C, A Modern Procedural Language .......... 41

3.3.3. The C++ Language .......... 46

CHAPTER 4. OBJECT ORIENTED DEVELOPMENT OF FINITE ELEMENT PROGRAMS

4.1. Introduction .......... 50

4.2. The Finite Element Method .......... 52

4.2.1. A Finite Element Displacement Formulation .......... 52

4.2.2. Current State of Teaching the Finite Element Method .......... 54

4.3. Object Oriented Development of Finite Element Programs .......... 56

4.3.1. General Guidelines for Object Oriented Development .......... 57

4.3.1.1. Levels of Abstraction .......... 58

4.3.1.2. Class Library and Reusability .......... 59

4.3.2. An Object Oriented Program for Finite Element Instruction .......... 63

4.3.2.1. Levels of Abstraction in a Finite Element Displacement Formulation .......... 64

4.3.2.2. Extension for Dynamic Analysis .......... 67

CHAPTER 5. AN OBJECT ORIENTED DATA MODEL FOR ENGINEERING DATABASES

5.1 Introduction .......... 75

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Review of Database and Data Models ........ 75

5.2.1. Basic Database Concepts and Terminology ........ 76

5.2.2. Data Models ........ 80

5.2.2.1 Network, Hierarchical, and Relational Models ........ 80

5.2.2.2 Data Model Issues for Engineering Data ........ 84

5.2.3. Object Oriented Databases ........ 85

5.3 An Object Oriented Data Model ........ 87

CHAPTER 6. SUMMARY AND CONCLUSIONS

6.1. Summary ........ 95

6.2. Recommendations for Further Research ........ 96

6.3. Concluding Remarks ........ 98

References ........ 99

Appendix: Declarations of Classes in OPFI ........ 109

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 1
# INTRODUCTION

## 1.1. Computer Integrated Structural Engineering

A broad definition of computer aided engineering is any support that the computer provides in delivering an engineered product. In structural engineering the product is a constructed facility such as a building, factory, bridge or dam [Fenves 89a].

The increase in use of computers in many aspects of engineering constructed facilities has been enormous during the past few decades. Computers are used for analyses of structures, design checks and optimizations, cost estimation, construction planning, producing engineering drawings, etc. Using proven algorithms, many of the individual tasks are highly automated; that is, they require very few directions during the solution process.

Engineers today approach new and larger problems with computers in mind as computational and information storage devices. Recently, powerful work stations have been procurable at reasonable prices and have dramatically increased the general availability of computers to a professional engineer. However, the increases in power and availability of computers have not changed the fundamental way computers are used in engineering constructed facilities; computers are still mainly used for computing many segregated but well defined algorithms and storing large amounts of pure data which have meaning only in the context of some application program. The methods of sharing data across organizational boundaries have not improved that much either; it is still common to find that in one office a designer uses a powerful computer aided design and drafting (CADD) package to produce a project drawing, and in another office, a construction estimator uses a digitizer to put the information from that same drawing back into another computer. The result is loss of efficiency and, perhaps more important, the potential for errors [Howard 89].



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

While competent human resources will become more valuable, computers will become even cheaper and more powerful. New computer software systems for engineering must be investigated with this premise. An environment that *integrates* many tasks is one such system. In this environment, an engineer can devote most of his time applying his creativity and abstract knowledge that are usually applicable to a set of tasks in a phase and not to individual tasks. For example, one needs creativity for good design and design involves many individual tasks such as analysis, serviceability checks, code conformance, aesthetic appeal considerations, etc; many of these individual tasks are already automated.

The term *integration* has been in fashion for some time in the computer aided design (CAD) field [Neeley 89] and the phrases *vertical integration*, *horizontal integration* and *levels of integration* are often used in the literature. In structural engineering, these phrases are used to mean the following [Sauce 89; Abdalla 89]:

1. Vertical integration refers to the integration of computers into all phases of engineering a constructed facility where the phases range from the early conceptual design to the final construction planning.
2. Horizontal integration refers to the integration of different tasks within a phase of engineering a constructed facility; for example, the tasks of analysis, serviceability checks, code conformance, and drawing generation in the design phase of engineering a constructed facility are brought together in a horizontal integration.
3. Levels of integration refers to the scales of integration; for example, a horizontal integration may be considered as low level (small scale) and a vertical integration may be considered as high level (large scale).

An integrated software system is still an objective and not a reality in structural engineering. Multilevel-Selection-Development Model for structural design process [Sauce 89], Component-Connection Model of buildings [Powell 88a], a model for functional and spa-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tial aspect of buildings in construction process [Garett 89], and other studies are conducted towards developing a fully integrated structural engineering system.

Interest in integration of programs is not new to engineering. Translators, neutral formats, and standard specifications have been used to passively integrate programs:

1. Translators: programs that take some data, generally an output from one module, and produce data in a format that can be read as input by another module. A translator is simply a data reformating program that integrates two modules. In civil engineering, the translator approach has been used to integrate pre/post processors with finite element analysis codes [Craine 81].

2. Neutral Formats and Standard Specifications: a module produces data in an agreed format (specification) and other modules read the information in that neutral format. A module only needs to read and write in a neutral format to become a part of an integrated system. IGES (Initial Graphic Exchange Specification) developed by the National Bureau of Standards is a defined format for the creation of a file which enables data found in today's commercially available CAD/CAM systems to be exchanged or archived [Smith 86].

A database can play a unifying role in an integrated system [Rasdorf 85], and by far the most appealing approach to develop an integrated engineering system is using a central database as the repository of information.

In traditional relational databases, the emphasis is on allowing multiple users to share a lot of the same accurate, consistent and up-to-date information. In the database for an integrated system for engineering, there is an additional emphasis that data must possess semantic richness as it is common in engineering to deal with complex data entities; in structural engineering, data are typically beams and columns and not integers and characters. Traditional relational databases are weak in handling complex information in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Computer Aided Design(CAD) and Computer Aided Engineering(CAE) [Dawson 89].

Many feel object oriented databases are suitable for dealing with complex design and engineering applications [Fenves 89b; Garett 89; Keirouz 87; Powell 88b]. The reasons why an object oriented integrated database system for structural engineering have not emerged yet are the following :

1. Object oriented data models for engineering are still under investigation.
2. Elegant models for tasks and processes in structural engineering that are to be integrated are still at theoretical stages.
3. The size of the project for any realistic problem is huge and requires long term commitment.

A software module in an integrated system must be versatile and flexible. The concepts of objects and object oriented paradigm in programming languages and software development represent the most promising unifying paradigm in the design and coding of a large software system. They have been used successfully to develop complex software systems in office information [Hogg 85], knowledge representation [Bobrow 83], and VLSI CAD [Katz 85]. Object oriented software development and programming languages show much promise in developing large engineering software in an integrated engineering system.

### 1.2. Objectives and Scope

The present study deals with the interface to the resources in a computer system that structural engineers need to face to develop integrated structural engineering systems. This includes a programming paradigm that is part of an integration tool and a data model that allows engineering views of contents in the database. An object oriented paradigm that may be found in programming languages and databases is the central and unifying theme in developing an integrated structural engineering system. The main objectives of this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

dissertation are concerned with an object oriented software development methodology and an object oriented data model that are appropriate for engineering systems; specifically, the objectives include the following:

1. Evaluate the advantages and disadvantages of using an object oriented language to develop engineering applications; identify a practical object oriented language that is appropriate to develop a large and integrated engineering software system. Then critically compare the object oriented language (C++) with languages that are currently used in engineering applications (Fortran and C).
2. Develop a C++ finite element program using object oriented concepts.
3. Based on the object oriented finite element program, devise a guideline for an object oriented development of engineering software where the programming paradigm is an essential tool in developing integrated engineering systems.
4. Develop an object oriented data model that is sufficient to support engineering entities in a central object oriented database of an integrated engineering system.

The scope of this dissertation is limited to investigating object oriented languages that are available to the public and object oriented database features that will likely be available in commercial object oriented database management systems. Implementation issues of language and database features are outside the scope of this study. In addition, impractical theoretical features are not studied. For example, new features in a language or a database management system are *not* investigated.

### 1.3. Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents object oriented concepts. The essential features of an object are outlined and how inheritance and polymorphism are utilized in object oriented approaches is discussed. Chapter 3

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reviews programming principles and programming paradigms. Fortran, C, and C++ are critically evaluated as a language for engineering software; the merits of C++ as a language for developing large and integrated engineering software systems are identified. Chapter 4 follows with a guideline for an object oriented development of engineering software; the guideline is based on a finite element program that is developed as a teaching tool using object oriented concepts and the C++ language. Chapter 5 reviews database systems and discusses the appropriateness of object oriented databases for integrated engineering systems. Requirements and description of an object oriented data model that will be viewed by engineering application developers and database designers are also presented. Chapter 6 summarizes the results of the study and presents some concluding remarks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 2

# OBJECT ORIENTED CONCEPTS

## 2.1. Introduction

This chapter summarizes some of the important concepts which are used in object oriented programming and data modeling.

The notion of an *object* as it is known today first appeared as a programming construct in Simula, a language for programming computer simulations [Birtwistle 71]. In the 1980s object oriented concepts were popularized mainly through the Smalltalk language and programming environment and many terms first used in Smalltalk are found in the literature to define and discuss object oriented concepts (See Chapter 3, Section 3.2.2.4, and references [Goldberg 83; Goldberg 84]).

Object oriented approaches are used in programming language constructs and mechanisms, data models, databases, execution-time support for objects in constructing object oriented applications, and environments for object oriented software development. The interest and promise of object oriented concepts have produced many definitions and interpretations by not only the computer scientists but also the general computer users. Although precise definitions are still not possible, more than a decade of history has produced general characteristics of object oriented concepts and techniques.

Basic object oriented concepts and terminology are presented in Section 2.2. Inheritance and polymorphism schemes are described in Section 2.3 as object oriented techniques. The focus in this chapter is the concepts and techniques that are relevant to object oriented programming languages and object oriented data models. Object oriented approaches that may be found in other applications are outside the scope of this dissertation.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 2.2. Basic Object Oriented Concepts

The terms used in this section are selected from variations found in texts on Smalltalk [Goldberg 84], LOOPS [Bobrow 83], CLOS [Keene 89], and C++ [Stroustrup 86].

### 2.2.1. Objects, Classes, Methods, and Messages

An object oriented system consists of many objects, and the system evolves as objects send messages to each other. Each *object* occupies a portion of memory and is an *instance* of a class. The values in memory constitute the *state* of the object.

A *class* is an abstract data type that defines the behavior and properties of objects that are *members* of the class. An object may be a member of many classes through inheritance (see the following section) but it can only be an instance of one class.

The *behavior* is the set of methods that are defined for its class. A *method* (called member function in C++) definition includes the following parts:

1. The generic function that the method specializes;
2. The method's applicability conditions;
3. Any qualifier that identifies the method's role;
4. A parameter list that receives the arguments; and
5. The body executed when the method is called.

The *property* of a class includes the data types and structure of object's memory which are called the object's *instance variables* (called data members in C++). Some data values may be the same for all objects in the same class and these data are associated with the class, not with each object. They are called *class data* (called static data members in C++).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The behavior (set of methods) and state (values in memory) comprise the *active* properties of an object. The data types and structure are sometimes called *passive* properties to distinguish them from these active properties.

A *message* is a syntactic construct that consists of:

1. An object identifier that singles out one object as the receiver of the message;
2. The message name; and
3. Some additional objects that are passed as arguments.

When an object receives a message, it invokes one of its methods. A special method may be designed to handle the messages that the object does not comprehend, i.e., does not have an appropriate method to invoke. The general result of a method invocation is a change in state, behavior, or property of the object. Although most applications limit this change to the state of the object, a messaging scheme where the behavior and properties of an object can be modified or extended is within the general object oriented concept.

### 2.2.2. Encapsulation

A means of dividing a large system into smaller and manageable subsystems that can easily be developed and maintained has always been a concern in computer science. In this respect, *encapsulation* is arguably the most important characteristic of objects and all object oriented approaches exploit encapsulation in various ways.

Data and classes packaged or *encapsulated* as objects are externally visible only through messages. The instance variables are visible only by the object's methods. The implementations of the methods are hidden and only the message interface is shown externally. The state, properties and behavior of objects are strictly managed by a set of methods defined by the object's class. In addition to modularity of encapsulated objects, the advantage is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that the rules for manipulating the object's data are defined when the object is defined and cannot be changed without changing the object.

Object encapsulation is fundamental in the development of a large integrated system, partly because the specification language for an object need not be the same as the implementation language. There is a great deal of software that is not written in terms of objects and this can be repackaged into objects with valid external interfaces; also, different languages may be appropriate or practical for implementing different kinds of objects while a common language can be used for external interfaces.

### 2.3. Object Oriented Techniques

Inheritance is often the fundamental feature that distinguishes object oriented approaches from others. Overloading and late binding are mere techniques that are much used in object oriented approaches to implement the concept of polymorphism.

#### 2.3.1. Inheritance

Class *inheritance*, or simply inheritance, is a basic reusability mechanism in an object oriented paradigm. The other basic reusability mechanism is the instantiation of objects via classes.

##### 2.3.1.1. Derived and Base Class Relationships

The idea behind inheritance is to provide simple and powerful mechanisms for defining new classes that inherit properties (i.e., methods and instance variables ) from existing classes. The new class is called a *derived class* and the classes that the properties are inherited from are called the *base classes.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The relationship between a derived class and its base class can be one of the following:

1. Specialization : This provides the basis for top down design of classes. The base class is a general class that captures the similarities between objects while the derived classes define the specifications for the differences.
2. Modification : This provides the mechanism for partial inheritance where only selected properties of the base class are inherited by the derived class; i.e., the set of properties that is not to be inherited can be *masked* or hidden to the derived classes.
3. Extension : This provides the mechanism for adding new properties by the derived class.
4. Aggregation : This provides the mechanism where a complex object can be constructed from a number of constituent objects, i.e., bottom up construction by providing a building block mechanism that allows complex objects to be built from previously defined simple ones. It also provides a basis for top down decomposition of complex objects into progressively simpler ones.

Note that the derived class is a subset (or sub type) of its base class *only* in the specialization type of inheritance. Instead of the terms derived class and base class, the terms *subclass* and *superclass* are often used in the literature; however, subclass and superclass imply the notion of subset and superset. The terms derived class and base class are more meaningful in the present context and are used subsequently throughout this dissertation.

#### 2.3.1.2. Single, Partial and Multiple Inheritance

The most simple inheritance is where a derived class inherits the entire properties of a *single* base class. But because the base class can be a derived class of another base class, a class in simple inheritance can have many base classes. The class hierarchy in a simple inheritance forms a *tree*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A *partial* inheritance is where there is a mechanism for inheriting only a part of the properties in the base class.

A *multiple* inheritance is where a derived class can have many base classes directly above it. Thus the class hierarchy where a multiple inheritance is allowed forms a *lattice* instead of a tree. Multiple inheritance is essential in many applications but a conflict of names for methods can be a problem. The system must provide default rules for selecting one method or for combining inherited methods, or the user must make an explicit choice, e.g., by preceding the method name with the base class name. The former approach can be error prone and the latter approach can become quite clumsy.

In order to preserve the encapsulation of the base, a derived class should not have *direct* access to the instance variables of the base class. However, many systems violate this rule for efficiency reasons (e.g., the *friend* mechanism in C++).

### 2.3.2. Polymorphism

*Polymorphism* is the ability to process a heterogeneous collection of objects in a uniform fashion. Inheritance, overloading and late binding are the techniques for polymorphism in an object oriented approach.

#### 2.3.2.1. Polymorphism via Inheritance

Inheritance is closely related to polymorphism: the property of a base class is exhibited by all the derived classes.

#### 2.3.2.2. Overloading

In a language with compile-time type declaration, an operator may be defined with the same name, but different argument types. When a call to that name is compiled, the com-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

piler uses compile-time type information to select which code to call. This is called *operator overloading*. When a function instead of an operator is overloaded, it is called an *overloaded function*. Operator overloading has existed in languages for quite some time. For example, the original Fortran did not allow mixed mode arithmetic; i.e., no operator overloading. The later versions of Fortran and many other programming languages allow arithmetic operations where arguments can be any arithmetic type and this is a form of operator overloading.

In an object oriented approach where there are many classes, which are similar to types in languages, operator overloading becomes a necessary interface to a user when there are many different classes with similar characteristics. Like an operator, which is just a syntactic convenience for a function calling scheme, overloading is a *syntactic* convenience that requires compile-time type checking of arguments. However, to a user of classes, an operator that maintains its behavior transparently for different arguments has found to be extremely useful. The limitation is that overloading only allows static polymorphism; i.e., the differences among objects must be known before execution time.

#### 2.3.2.3. Late Binding

In theory of programming languages, *binding* is usually understood as the process of matching compiled binary modules to produce an executable image which involves assigning a memory address to each module and patching external references with the correct memory addresses. When binding occurs during compilation and before execution time, it is called *early* binding (or static binding) to differentiate it from *late* binding (dynamic binding). Late binding means that the binding occurs at execution time.

Binding in an object oriented system concerns the selection of method to respond to a message: in late binding, the method selected to respond to a message is determined at execution time. Late binding in an object oriented paradigm is the only way to uniformly treat

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

similar objects whose differences are determined at execution time, and this allows more flexible run time polymorphism.

Late binding is also important for flexible behavior of objects where the objects are allowed to receive any message during execution time; however, this flexibility can cause more errors since compile-time checking is impossible.

Concepts described in this chapter are used subsequently to describe an object oriented programming development environment and data modeling suitable for structural engineering applications. Prior to this discussion, alternative programming languages are first considered in Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 3

# PROGRAMMING LANGUAGES FOR ENGINEERING SOFTWARE

## 3.1. Introduction

Engineering activities make heavy use of computers and with computer technologies evolving rapidly, much of the challenging computer aided engineering activity involves writing new programs in addition to running existing ones. The selection of a programming language is one of the initial decisions that influences the quality of software, acceptance by users, and cost of development and maintenance. In the past, engineering programmers were relieved of making this decision because Fortran was *the* programming language for most engineering software.

Fortran still remains an effective language to write programs when the major complexity in the problem is numeric computations. Today, the size of problems and engineering software systems have increased dramatically and numeric computation is only a part of problem; major problems (e.g., the so called *software crises*) are in control of interactions among modules, management of data, and integration of software systems. Traditionally trained engineers are competent in handling numeric complexity but they lack computer fundamentals to attack the software crisis. One of these fundamentals is programming principles. The rest of this chapter reviews programming principles and presents C++ as a language that supports current programming paradigms and is suitable to meet the challenges of today's engineering software crises.

This chapter is organized as follows. Programming styles and paradigms are presented under Programming Principles in Section 3.2. Procedural, data hiding, data abstraction and object oriented paradigms are reviewed. Section 3.3 discusses languages for



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

engineering software in general and Fortran, C and C++ in particular. Deficiencies in Fortran as a procedural language are elaborated. C is discussed as a modern procedural language and C++ is presented as a superset of C in which the added supports for the object oriented paradigm blend well with the base language C. The C++ language is proposed as an appropriate language to develop today's engineering application programs.

### 3.2. Programming Principles

Programming principles give unified and logical approach to program design and code whose neglect will prove disastrous for *large* projects. Programs based on the maxim "First make the program work, then make it pretty" may be effective for small programs but simply will not work for large ones [Kruse 84].

Programming principles are one of the earliest fields established in computer science. Programming styles and programming paradigms are essential parts of programming principles.

#### 3.2.1. Programming Styles

A disciplined programming style makes debugging and maintenance of large programs possible. No large program is bug-free and all programs need *maintenance*, i.e., the need to meet new user requirements and to operate in changing computer environments.

Generally, development of software systems follows these stages [Sommerville 89].

1. Specify Requirements : Precisely formulate and specify the software requirements. This results in a formal document commonly called SRS (Software Requirements Specification) in the industry and it becomes a part of the contract between the client and the software developer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Design : Selecting algorithms, organizing data structures and coordinating personnel are all parts of software design.
3. Coding : Coding is the translation of the design into codes.
4. Testing : A set of tests are conducted so that the product meets the requirements in SRS and the developer's own standards.
5. Delivery : The code is delivered to the client and the client undertakes a set of tests before accepting the product.

An undisciplined programming style usually places too much emphasis in coding. Theoretically, coding is just a translation of instructions from the design stage into legitimate codes. When the instructions from the design, commonly in pseudo code, are clear and logical, coding is straight forward. However, errors and poor design are often detected during the coding stage. Fixing a minor error in design becomes a major task. Poor design produces poor product and in unfortunate cases, total failure.

The life cycle of an engineering software system varies from one run (sometimes unsuccessful) to about five years. Maintenance follows the initial development. The cost of maintenance is estimated somewhere between one-half [Kruse 84] to two [Bell 87] times the cost of development. Whatever the exact figure is, this cost is usually underestimated, and sometimes forgotten or neglected during the development stages.

For example, the U. S. Government Accounting Office (FGMSD-80-4) reported in 1980 the following breakdown of federal software projects :

| | |
|---|---|
| 3.2M (47%) | - paid for but not delivered |
| 2.0M (29%) | - delivered but not used |
| 1.3M (19%) | - abandoned or reworked |
| 0.3M ( 5%) | - used after rework or as delivered |

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Almost 95% were failures and the reasons can be attributed to changing user requirements and unreliable codes that are impossible to maintain.

A *reliable* or *robust* code is a code that can be maintained. A simple bug can be found and fixed without introducing a new one. Minor additions and modifications can be made without a major reorganization. Reliable codes are characterized by

1. Clarity
2. Modularity and
3. Structured Programming.

*Clarity* applies to the executable source code but also to comments and external documents. In the code, mnemonic names are used for variables and function names. Familiar algorithms are used instead of cryptic processes with tricks. Clarity often takes precedence over efficiency in computation or data storage. The lay out of the code on the screen is organized using indentation and spacing so that relations among the elements of the code are clear to the viewer.

*Modularity* characterizes the independence of modules. A module can be a block (group of statements), a procedure, or a group of procedures in a single file. A module's function must be general so that it can be widely re-used. The independence of a module depends on how the module is coupled with the rest of the program. Cassel identifies three types of couplings between modules and they are listed below in order of decreased independence [Cassel 83]:

1. Data coupling : Data coupling occurs when only data is passed or shared between the modules. Modules coupled by data are considered independent.
2. Control coupling : Control coupling occurs when flags are passed and an operation in one module affects an operation in another. Control coupling can be removed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by redesign or by dividing the module into smaller independent modules. This type of coupling is acceptable if it is not too extensive. Extensive control coupling violates structured programming.

3. Content coupling : Content coupling occurs when one module alters the contents of another module. This is a severe coupling between modules and the name module is a misnomer in this case.

*Structured programming* is now intuitive to most programmers. It is based on the proof that all computer programs can be coded by using only three logic structures or combinations of these structures [Dijkstra 76] :

1. Sequence : Sequential execution of statements.
2. Selection : IF-THEN-ELSE and CASE (or SWITCH) types of statements.
3. Repetition : FOR, DO-WHILE and REPEAT-UNTIL types of loop statements.

The three structures are useful in a disciplined style of programming because the code is simplified. Only the three building blocks are used and as a result there is a single point of entry into the structure as well as a single point of exit. Structured programming enables the program to be read top to bottom, making the logic of the program visible and understandable to those concerned with debugging and maintenance. A go-to statement is a culprit in structured programming and there are two schools of thought: one advocating "no go-to" and the other advocating "limited go-to". Use of go-to for error handling is generally considered acceptable [Knuth 74].

### 3.2.2. Programming Paradigms

A programming *paradigm* is a form of solution to a problem and it guides the approach to arriving at a solution. Four paradigms starting with the oldest to the most recent one are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

listed below. Representative programming languages that support each paradigm are also shown below with approximate year of initial implementation.

1. Procedural : Fortran (1957), Pascal (1971), C (1972)
2. Data Hiding : Modula-2 (1979)
3. Data Abstraction : Ada (1980s)
4. Object Oriented : Smalltalk-80 (1982), C++ (1983), Objective-C (1984)

These paradigms offer different *models of abstraction* that systematically divide and conquer complexity that are inherent in large problems. The four paradigms are discussed next in terms of language supports that are necessary to support each paradigm. Features in the languages listed above are used as examples; the syntactic details of specific languages are omitted so that general ideas are highlighted.

#### 3.2.2.1. Procedural Paradigm

The procedural paradigm is the oldest and most common programming model. It is also called functional decomposition because the idea is to divide the problem into functional components and focus on operations and algorithms needed to perform the desired computations in each function. Reliable parameter passing schemes between functions and expressive operators in a language are necessary to support a procedural paradigm.

There are many ways of parameter passing and an abridged list is shown below [Rowe 86]:

1. Call-by-value : Parameter value is passed and the value is discarded upon exit from called function.
2. Call-by-value/result : Parameter value is passed and the value is copied to a passing parameter upon exit from the called function.
3. Call-by-copy-in/copy-out : Similar to call-by-value/result, but the passing parame-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ter is evaluated at copy out.

4. Call-by-reference : This is formal reference to a calling parameter which is passed and returned upon exit. This scheme can cause bad side effects.

5. Call-by-name : This scheme evaluates the parameter in the calling environment every time it is referenced.

Only call-by-reference and call-by-value are found in most languages. Fortran is call by reference, C is call by value, and Pascal supports both call by value and reference (called variable parameter in Pascal).

In programs written in languages that only support a procedural paradigm, the operations and data are intertwined and it is hard to distinguish the algorithm from the data. This lack of attention to data design and data structure in a procedural paradigm yields fine granularity, i.e., small building blocks. We find that procedural programs are difficult to reuse. Often a procedure's function meets the requirements but the procedure cannot be used as it is and modification generally entails comprehension of many unnecessary details. Fortran is the original procedural programming language; Pascal and C came later in the same tradition but they give some attention to data structure: *records* in Pascal and *structures* in C [Davis 78; Wirth 71; Kernighan 87].

As problems became larger over the years, it was recognized that complexities cannot be resolved by procedural paradigm alone and emphasis was shifted to organization of data. The next three paradigms - data hiding, data abstraction, and object oriented - show progressive advancement and elegance in organization and the use of data. Even programming of complex algorithmic problems may be greatly simplified by use of well designed data structures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

#### 3.2.2.2. Data Hiding Paradigm

The data hiding paradigm is also called modular programming. The idea is to group data and implementation details in modules and provide a clear interface for each module. When a module is implemented, only the interface is shown to users.

Modula-2, a language that supports data hiding, is a descendent of Pascal and Modula. The syntax is close to Modula but Modula-2 is essentially an extension of Pascal with module concepts. The interface part (called definition part in Modula-2) and the implementation part of a module are clearly separated. Other language supports for data hiding come in explicit control of the scopes of names (import/export) and module initialization mechanisms [Wirth 88].

Fortran and Pascal do not have any features that support data hiding. In C, data hiding can be implemented by grouping related data and implementation into a single source file and then a separate header file can be shown as the interface. This, however, does not necessarily mean that the C language supports data hiding. A language supports a paradigm or style if the language provides facilities that make it convenient and safe to use. For example, one can write structured programs in Fortran and use data abstractions in Modula-2, but these languages do not support these techniques [Stroustrup 88a].

The separation of implementation and interface in data hiding is similar to a black box concept. This is the concept of encapsulation found in object oriented programming and it results in codes that are easier to maintain and re-use than procedural programs. The interface of a module is the external description of the black box and it is analogous to the description of the built-in type of a language. However, the use and behavior of a built-in type is much more convenient than a module. The syntax for use of a built-in type is simple and multiple uses of it do not require special handling.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

#### 3.2.2.3. Data Abstraction Paradigm

The phrase *abstract data type* is used in the literature to mean a user-defined type and it constitutes the concept of class in object oriented programming. The objective of a data abstraction paradigm is to have user-defined types behave and be used exactly like built-in types. Data abstraction is an essential subset of an object oriented paradigm and often is confused as *the* object oriented paradigm. To highlight the essential relationship and the difference between the two paradigms, the data abstraction paradigm is also called the *object based* paradigm.

To support data abstraction, a language provides mechanisms to define a full set of operations and hide the implementation details for user-defined types. The concept of hiding the implementation details is from the data hiding paradigm. The mechanisms to define a full set of operations may be divided into four categories:

1. Initialization (constructors) and declaration.
2. Implicit conversion.
3. Operator overloading.
4. Destructors.

The mechanisms for initialization of user-defined types must allow unrestricted use of assignment (=) and other declaration formats that are used for built-in types. The mechanisms for implicit conversion, which are internally related to the initialization mechanisms, are also necessary for convenient use of existing operators. For example, if an operator is defined for one type of argument and the conversion from the actual type of argument to the type required by the operator is obvious, then this should be allowed by mechanisms in the language for implicit conversion. Implicit conversion also enables more flexible use of operator overloading, which is an essential part of data abstraction paradigm. It minimizes the distinction between built-in and user-defined types. Destructors are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operations to free the memory when objects of user-defined types are no longer used.

Support for a data abstraction paradigm is at the expense of computer efficiency both at compile and execution time. Many of the mechanisms rely on additional function calls and there is always an overhead associated with a function call. The overhead for operator overloading is that the argument types have to be looked up before determining which code to execute. Codes that are packaged as user-defined types are more general and reliable than modules in data hiding.

A type, whether built-in or user-defined, is a concrete representation of a concept. The reason for designing a new type is to provide a specific definition of a concept that has no direct and obvious counterpart among the built-in types. Many of the pioneering development and implementation of user-defined types are found in the languages Simula [Birtwistle 71] and CLU[Liskov 77]. Simula is a popular language in Scandinavia and CLU is an experimental language developed at MIT. Convenience and logic in using and defining user-defined types can be accomplished with elegant definition of a language. Implementing the language so that execution and compile time behaviors are acceptable is perhaps a more difficult problem; for example, balancing the extent of type and array bound checking at compile time to ensure reliable behavior at execution time is a difficult problem.

Ada is a general purpose programming language commissioned by the US Department of Defense to replace the diverse collection of languages that are used to build their computer systems. It is a complicated language that incorporates many desirable features, and supposedly has input from most language design experts in the world. Ada supports the data abstraction paradigm with the concept of a *package*. A package in Ada contains parts that are private. This is the part that the package developer does not want the users to see. The users only sees the interface part of the package which the package developer has chosen to show. Ada supports operator overloading so that the use of packages can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more flexible. Initially, the benchmark tests for execution time and compile time behaviors of a small portion of Ada code were disappointing and many concluded that Ada would never become a practical language. However, recent reports on Ada benchmark tests show marked improvements, comparing reasonably well with codes written in Fortran or C [Ichbiah 79; Ada 83; Ada Letters 1990].

#### 3.2.2.4. Object Oriented Paradigm

The object oriented paradigm extends the data abstraction paradigm in ways in which the classes may be defined and the objects may be operated. The Object oriented paradigm allows classes to be more general and flexible than the ones based only on a data abstraction paradigm by using two techniques (See Chapter 2, Section 2.3.):

1. Inheritance; and
2. Polymorphism.

In the object oriented paradigm, classes are organized as a set of inheritance hierarchies and the relationship between a derived class and a base class may be one of a specialization, modification, extension, aggregation or combinations of these as discussed in Chapter 2, Section 2.3.1.1. A class can also be derived from more than one base class, i.e., multiple inheritance. The essence of an object oriented paradigm is not just in defining classes, but it is in how flexibly the classes can be derived from existing classes and how general the classes can be defined so that they become widely re-usable base classes.

Ada is somctimes said to support an object oriented paradigm because it allows one to derive new types from existing types. However, a new type derived from an existing type in Ada is not really a new type because Ada only allows restrictions on an existing type to derive a new type; for example, if there is an existing integer type that can hold values from 1 to 100, a new integer type may be derived from this type where the new type can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

hold any subset of integers from 1 to 100, but any new type derived from this integer type cannot be programmed to have integers less than 1 or greater than 100. These restrictions on how a new type may be derived from an existing type are severe and Ada cannot be said to support an object oriented paradigm.

The oldest example in the literature on object oriented programming uses *shape* where an object in that class responds to the message *draw*. The derived classes are circle, square and triangle. When message *draw* is sent to a shape object, the method selected to respond to this message is determined at execution time and it depends on whether the object is a circle, square or triangle. At a later time, when a new shape, say an ellipse, needs to be added, the programmer merely defines ellipse as a derived class of shape and provides a method to draw it. The important implication of this simple example is that late binding combined with inheritance allows execution time polymorphism.

One of the most important objectives of programming styles and paradigms is to produce programs that can be maintained. Developing large systems are expensive. This cost is justified only if the developed system can be maintained during its expected life cycle. The three previous programming paradigms - procedural, data hiding, and data abstraction paradigms - assume the need to modify existing codes to meet new user requirements and to operate in changing computer environments. An object oriented paradigm is fundamentally new in its approach to maintenance in that execution time polymorphism can be used to modify or extend capabilities of a system *without* modifying existing codes.

*Object oriented programming languages :*

Data hiding and data abstraction paradigms are essential parts of object oriented paradigm. The complete language supports for an object oriented paradigm *must* include the following *four* features:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Encapsulation (from the data hiding paradigm).
2. User-defined type (from the data abstraction paradigm).
3. Inheritance.
4. Polymorphism.

The language that has probably had the most influence on the evolution of the object oriented paradigm is Smalltalk [Goldberg 83; Goldberg 84] and it may be considered as the ancestor of all object oriented languages. Other languages that support the object oriented paradigm include C++ [Stroustrup 86], Objective-C [Cox 86], LOOPS [Bobrow 83], Flavors [Cannon 80], CLOS [Keene 89] and Object-Pascal [Schmucker 86]. Smalltalk is a pure object oriented language but the other languages are hybrid languages that are extensions of some base language. C++ and Objective-C are extensions of the C language. LOOPS, Flavors and CLOS are LISP based languages that extend the functional approach to problem solving by emphasizing class hierarchies to organize facts about a problem domain. Object-Pascal is Apple's object oriented extension of Pascal where the syntax was partly designed by Nicklaus Wirth, the designer of Pascal. Smalltalk, C++, and Objective-C are three widely used object oriented languages and they are discussed in this Section. Other hybrid languages mentioned above are not apt as languages to develop large systems because their base languages are not considered as general purpose languages. LISP is used mostly in artificial intelligence applications. Pascal was designed as a teaching tool and its lack of support for separate compilation severely limits its capability to develop large programs. A more detailed survey of object oriented languages and their uses is given in the book by Schmucker, *Object-oriented Languages on the Macintosh* [Schmucker 86].

The three language extensions necessary to support object oriented programming are the syntaxes for the following:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Defining classes, including inheritance.

2. Instancing objects.

3. Sending messages to objects and late binding mechanism.

*Smalltalk :*

Smalltalk only has syntax for the three extensions mentioned above and this results in a simple and small language. All binding occurs during execution and the language is made even smaller and consistent by eliminating any concern for types; everything in Smalltalk is inherited from only one type: an object.

The programming language aspect of Smalltalk is discussed here but Smalltalk is best described as a programming environment that integrates the programming language, the operating system, and other support tools such as editors, linkers and debuggers. The entire environment is a research product of Software Concepts Group at Xerox PARC and from the beginning its purpose has been to prove concepts. The general concepts had remarkable influence on many academic and commercial systems such as Xerox's Star office automation system, Apple's Lisa, and many of today's high-resolution graphic workstations. Smalltalk-80 is a version of the language, and it was purely a secondary product. Smalltalk does not have any standard for the language or the class libraries and it is not expected that there will be any portability among different versions of Smalltalk. The language described in *Smalltalk-80: The Language and its Implementation* [Goldberg 83] and the June 1981 *BYTE* magazine issue about Smalltalk are generally used in the industry to derive different version of Smalltalk.

There is elegance in simplicity provided by a *pure* object oriented language like Smalltalk-80. The traditional way of developing a program - start with an editor followed by compile-debug cycle - may be used to code in Smalltalk but the Smalltalk environment was designed for a completely different approach to program development. The Smalltalk

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

environment is composed of interacting objects that are instances of classes. A new code is tested by running it directly in the environment, which includes an operating system, rather than compiling it separately and then running the compiled code under the control of the operating system. Programming in Smalltalk proceeds by expanding the environment, not by generating a separate body of source code that is the product of traditional programming. In order to be effective, familiarity with the tools in the Smalltalk environment is as important as the knowledge of object oriented programming. For example, the *browser* and *inspector* classes are essential tools in the Smalltalk environment. Typical objects in the *browser* classes are used to browse classes and class hierarchies in the environment. When a class is selected using a browser, the text defining the class may be brought to the screen. The class can be modified and when the new definition is saved, the new definition is compiled.

The object oriented paradigm and the tools provided in the Smalltalk environment are new to most programmers, and both must be utilized. Smalltalk language's lack of support for other programming paradigms and its departure from the traditional way of program development have made it difficult for experienced engineering program developers to adopt Smalltalk. From a technical perspective, the execution behavior of Smalltalk programs are still not very efficient. Late binding and operator overloading are useful, but they have execution time overhead. Treating everything as objects yields consistency but long inheritance hierarchies causes difficulty in depicting full properties of an object, especially to novice object oriented programmers and those who are not proficient in using tools such as browsers in the Smalltalk environment. Smalltalk vendors have targeted as their clientele professional programmers such as those building commercial Macintosh and OS/2 applications for current and future products [Bergman 90].

Many extensions in hybrid languages compromise support to the object oriented paradigm for execution time efficiency. Currently, a typical conventional program written in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fortran, C, or Pascal is expected to execute faster than an object oriented program. The main reasons for execution time inefficiency in object oriented programs are late binding, argument type checks for operator overloading and extra function calls. The Smalltalk language ignores this problem and hopes that more efficient implementation and optimizers will be developed.

*Objective-C and C++ :*

Hybrid languages C++ and Objective-C are both based on C. With the exception of minor details in C++, they both keep C as a subset. Therefore the programming paradigms supported by C are retained. Binding is either early or late in both C++ and Objective-C. When there is no need for late binding or the object oriented paradigm, the conventional C features make these languages more efficient than Smalltalk-80. One feature that is missing altogether in Objective-C is operator overloading. Usefulness of multiple inheritance is recognized in Objective-C, C++ and Smalltalk-80. Original Smalltalk did not support multiple inheritance but some later versions support it. In Objective-C, multiple inheritance is not supported directly but can be simulated with a coding trick (see pg 90, [Cox 86]). No such trick existed for C++ and multiple inheritance was not supported in the original version of C++; however, later versions of C++ provide support as proposed by Stroustrup in [Stroustrup 87].

Both Objective-C and C++ provide the three language extensions stated above that are necessary to support an object oriented paradigm. These extensions are conceptual but a language is more than just a support for concepts. Perhaps more important is the elegance and uniformity in the syntax of the entire language. Objective-C adds just enough capabilities to support an object oriented paradigm. C++ redefines the C language to enhance the procedural paradigm of C and to support an object oriented paradigm. The approaches used by Objective-C and C++ to extend the base C language represent two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

fundamentally different ways in how a hybrid object oriented language may be defined.

Objective-C is one of the first object oriented languages that was developed as a commercial product. It was originally sold by Productivity Products International and its founder, Brad Cox, is the author of one of the earliest texts on object oriented programming and Objective-C [Cox 86]. One of the concerns in designing Objective-C was portability and the extensions were designed so that a simple translation of Objective-C to equivalent C source code was possible. Objective-C adds just one new type and one new operation to C. The new type is the object and it is called *id*. The new operation is the message expression used to send messages to objects and has this syntax:

```
_msg( object-id, "MESSAGE", argument1, ... )
```

where `object-id` is type *id* and `MESSAGE` is the string that contains the message. All objects in Objective-C are of type *id* and messages take the same form. Binding of methods to messages and distinction between different kinds of objects are handled at execution time. These simple extensions are designed to support the Smalltalk-80 style of object oriented programming. A deficiency in Objective-C as a general programming language is that the object oriented extensions do not blend with the C language; this was never intended and some may not consider it a problem. However, uniform treatment of many different features is a prominent characteristic of a powerful general purpose language.

C++ [Stroustrup 86] is an enhancement to the C language where one of the major enhancements is support for the object oriented paradigm. A new keyword introduced in C++ is *class* and is used to define new types. However, *class* blends well with the rest of the C language because the concepts behind it are not entirely new; *class* is *struct* in C extended to support user-defined types and inheritance. Once a class is defined, object instantiations follow similar forms that are used in C to declare objects of built-in types. For efficiency

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reasons, late binding is limited to those methods that are explicitly declared as *virtual* in the base class and other methods that are all bound at compile time. The C++ language is an efficient language that uniformly supports many programming paradigms, one of which was the object oriented paradigm. In Section 3.3.3, C++ is discussed in more detail.

#### 3.2.2.5. Other approaches to programming

This discussion on programming will not be complete without mentioning functional and logic programming. These are so called *declarative* approaches that depart radically from the mainstream of programming paradigms discussed thus far.

*Logic* programming aims at supporting programming instructions that are easy for humans to provide. This contrasts dramatically with conventional instructions based on data and operations which are easy for a computer to understand. Logic programming suggests that explicit instructions for operations not be given but the knowledge about the problem and assumptions that are sufficient to solve it be stated explicitly as logical axioms. The program is executed by providing it with a problem, formalized as a logical statement to be proved, called a goal statement. The execution is an attempt to prove the goal statement given the assumptions in the logic program. Prolog is the prime example of a logic language. The origins of the language can be traced to early 1970s to Alain Colmerauer and his colleagues at the University of Marseille-Aix but a major boost to Prolog and logic programming came in October 1981 with the announcement of the Japanese Fifth Generation Project where they adopted Prolog as the programming language. [Colmerauer 73; Sterling 86].

*Functional* programming is based entirely on functions which are considered as values that do not change. In functional languages, all procedures are functions of their arguments

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which are bound at execution time. There is no provision for assignment or mutable data. The assignment is an essential operation in conventional programming but functional programming advocates say that assignment complicates reasoning about programs because it introduces time boundaries into process; the value of a variable is changed by an assignment at the moment of the assignment, which makes uses of that variable before the assignment different from uses of that variable after the assignment. This actually is one of the fundamental reasons why conventional programming is sequential and devising a concurrent language based on the conventional programming model is hopeless.

The most common functional language is LISP [Wilensky 84], a language known for many dialects which includes MacLISP (developed at MIT), InterLISP (partly developed at Xerox PARC) and FranzLISP (developed at UC Berkeley). Portability of LISP programs is a concern because of many dialects and Common LISP is a version that tries to set a more uniform standard for LISP programs. LISP is an acronym for LISt Processing. A *list* is a binary structure where the first argument holds an element and the second argument is recursively the rest of the list. Intuitively, a list is items enclosed in parenthesis where an item can be atomic (nothing or an element) or another list. The syntax of the LISP language is just lists where the first item is the function name and the rest are arguments. LISP is normally used as an interpreter where a programmer enters lists and LISP evaluates them. The LISP language was first conceived by John MacCarthy and his students at MIT in the late 1950s. LISP is one of the oldest languages that has not only survived but also flourished in the field of artificial intelligence. More modern functional languages are Id [Nikhil 88] and ML [Milner 84]. Baugh demonstrates applications of functional programming to develop finite element systems [Baugh 89].

Functional programming and logic programming are used mostly in artificial intelligence applications such as expert systems and natural language processing. Another dominant area of interest in these programming approaches is their parallelism. The promise of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel computers and parallelism that seems to be available in logic programming and functional programming have lead attempts to devise concurrent languages based on these programming models.

### 3.3. Programming Languages for Engineering Software

Programming Languages provide tools to implement the paradigms and which paradigm to use often depends on the availability of tools. The language also governs the way we think about the problem and the details that actually make a solution work. The distinction between a *support* for a paradigm (or style) and a mere *enabling* of a paradigm is important. A language supports a paradigm if it provides tools that make it convenient to use that paradigm. If it takes exceptional effort and tricks to implement a paradigm, the paradigm is not supported although we may say that it enables the paradigm [Stroustrup 88]. A paradigm that is not supported but enabled in the language by coding tricks often results in cryptic codes that are sources of errors during the maintenance cycle.

Literally hundreds of computer programming languages are defined; however, most engineering applications software are written in Fortran. Pascal is a popular teaching language in many engineering institutions but is not used to develop large systems because the language does not support separate compilation of its units, i.e., when any portion of the code is altered, the entire program has to be recompiled. Fortran and C support separate compilation which permits the programmer to compile pieces (functions and subroutines) separately and later paste them together during linking. The C language has gained some acceptance by engineers. Although Fortran and C are both procedural languages, there are distinct features of C that make it a modern procedural language. Recently, C++ has become arguably the most widely used object oriented language. C++ has C as a subset so it is also a procedural language.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This section discusses the following:

1. What exactly is deficient in Fortran.
2. Features in C that make it a modern procedural language.
3. Procedural enhancements and support for the object oriented paradigm in C++.

The discussion assumes some familiarity with Fortran, C and C++. Evaluating a language simply based on whether a feature exists or not is deceiving because a new feature can always be added as it is demonstrated by Fortran. Thus the following discussion does not compare Fortran, C, and C++ based on a set of specific criteria, but each language is discussed separately in terms of its general features.

A procedural paradigm alone is not sufficient to develop a complex engineering system. The need to adopt a new language that supports modern programming paradigms which may be used to develop complex engineering systems is apparent: C++ is proposed to be that language. For almost forty years, engineers basically settled with Fortran. Not all were content with Fortran as there were attempts in the past to change the old habits and adopt a new language such as PL/I [Augensten 79] or Algol [Anderson 64]. Fortran not only survived but even today dominates development of new software. More than four hundred years ago, Nicolo Machiavelli understood man's reluctance to change his tools as he wrote in *The Prince*:

> Nothing is more difficult to carry out, nor more doubtful of success,
> nor more dangerous to handle, than to initiate a new order of things.
> For the reformer has enemies in all those who profit by the old order,
> and only lukewarmness arising partly from ...
> the incredulity of mankind, who do not truly believe in anything new
> until they have had actual experience in it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 3.3.1. Deficiencies in Fortran

Fortran, which is an acronym for FORmula TRANSlator, was originally conceived in 1953 by Backus and his colleagues at IBM [Backus 81]. The language was not really designed because the primary purpose at the time was to evaluate science and engineering formulas as efficiently as possible. The first version of Fortran was released in 1957. Fortran-II was released in the following year and it corrected some shortcomings of the original version. The short-lived Fortran-III introduced boolean and alphanumeric data. Fortran-IV made a few changes in the basic instructions and added some additional features to Fortran-III. Fortran-IV eventually evolved into Fortran-66, which was the first official standard. A revised American National Standard Fortran was adopted in 1978 and this version, Fortran-77, is available for use on almost all computers today. Fortran-77 is the second standard, but it is the first standard designed by a standards committee. Major new additions in Fortran-77 were in connection with file processing and character manipulation. Fortran-77 also clarified some ambiguities in Fortran-66, especially in the DO-loop parameters. Other versions, namely WATFOR, WATFIV and WATFIV-S, were developed at the University of Waterloo as teaching tools and some of their significant features were adopted in Fortran-77 [Davies 78].

Computer scientists and programming language experts have concluded long ago that Fortran is deficient with many features that are obsolete. The reason for pervasive use of Fortran in engineering applications today is believed to be the *inertia* (billions of lines of Fortran code exist) of Fortran codes and not the inherent advantages in the language [Fenves 89a]. Even the initial popularity was not because of the merits in the language but because it was offered and supported by the largest hardware supplier, IBM [Holtz 88]. However, inertia and long history have attracted brilliant minds from time to time and they have produced very efficient Fortran support tools, such as compilers, optimizers and mathematical libraries, that aid in producing efficient engineering and science application programs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This is a reality and should not be taken lightly; enormous amounts of efficient Fortran codes exist and any engineering system that requires rewriting of these codes is not economically feasible. Only new codes should not be produced using obsolete methods and new minds should not be limited to the knowledge of Fortran.

Reliable codes are characterized by clarity, modularity and structured programming (see Section 3.2.1). Fortran does not support clear coding, modularity, or structured programming.

Clear coding includes giving mnemonic names to variables, informative comments and visually organized layout. Fortran variable names are restricted to six characters or less, case (upper/lower) is insignificant and comments have to start in column 1. The source code is fixed format (i.e., label in columns 1-5, continuation in column 6, statements in columns 7-72, C in column 1 for comment) which is a remnant from card punching days. These restrictions are considered obsolete and do not exist in other languages. Some Fortran compilers allow more than six characters for variable names but this allowance can cause nightmares since some other compiler may ignore the characters beyond the sixth; for example, COLUMN1 and COLUMN2 can end up referring to the same variable COLUMN.

Another obsolete Fortran feature is the treatment of blanks. Blanks in Fortran source code are treated as insignificant, except in FORMAT and in character strings. Compilation involves one or several passes over the source code by scanner, parser, semantic analyzer, and object code generator. The scanner transforms the source code into a string of tokens where a token is an indivisible element of the source code. Scanner is a simple one pass operation with a minimal or no look ahead for most other languages. Because blanks are insignificant, the scanner in a Fortran compiler requires analysis and back tracking. For example, the scanner cannot conclude whether the statement DO 10 I=5,20 is a DO-loop or an assignment of 5 to variable DO10I until the comma is processed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A program unit in Fortran is the procedure. Fortran does not provide nested procedures (procedure defined inside another procedure) or recursive procedures (procedure that calls itself). Fortran has two kinds of procedures: functions and subroutines. Such distinction is actually not necessary since a function is a subroutine that returns a value or a subroutine is just a function that does not return a value. The interactions among subroutines in Fortran programs are usually through a *long* list of arguments and data in the common blocks. In a typical Fortran program, there is no modularity among procedures as huge common blocks are accessible by most subroutines and long lists of arguments generally depict some serious coupling among subroutines. The language does not support modularity, but Fortran *users* have aggravated the problem by devising coding methods that enable programming features that are not supported by the language. Engineering programmers in Fortran have for years used array indexing as an inelegant and nonstandard form of pointers to a large array in a common block. The addresses of an array are used to control the sizes of sub arrays within the large array and some have called this dynamic storage. Memory management has always been a major concern in engineering programs and if solving the largest problem solvable by the computer is a primary concern, this method works. Furthermore, equivalencing arrays of different types and making use of the varying byte-sizes of the types, a typeless pointer is concocted. These are programming tricks to simulate a concept that is not supported in the language. Pointers are actual memory identifications and dynamic storage is storage that can be allocated or deallocated at execution time; Fortran does not have pointers and Fortran does not support dynamic storage.

Parameter passing in Fortran is call-by-reference. Call-by-reference can cause bad side effects because a foreign environment, the called subroutine, has access to data that belongs to the calling environment. One of the problems in parameter passing in Fortran is not because it is call-by-reference, but because of *misuse* of call-by-reference by engineering programmers. In many engineering programs, an integer parameter is used as a con-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

trol flag, i.e., depending on the value of the parameter, the called subroutine branches to different cases inside the subroutine. The problem is that a constant is often used as the actual parameter. Consider a simple example where there are two subroutines, A and B.

*file 1 :*
```
SUBROUTINE A
CALL B(1)
```

*file 2 :*
```
SUBROUTINE B(N)
N = 2
```

A statement in A calls B with constant 1. Subroutines A and B will compile, but what is going to happen when the program is loaded and executed? With some old compilers, 1 mysteriously became 2. Fortunately, this problem is known to compiler writers but the error can only be detected during execution time. Theoretically, call-by-reference should *not* allow constants as actual parameters but such a requirement will not be popular since use of constants as actual parameters in Fortran is common.

Fortran does not have any features that support structured programming. One of the primary language supports for structured programming is a *block* structure. A block structure is a group of statements where there may be variables local to the group and where the group may be placed anywhere a single statement is allowed. Block structures in other languages usually have the following syntaxes:

1. BEGIN ... group of statements ... END
2. { ... group of statements ... }

Although structured programs can be written in Fortran, computed GOTO's and lack of block structures in the language have yielded many *un*structured Fortran programs in engineering.

The most touted feature of Fortran is the rich set of intrinsic numerical routines. The reliable behavior of many of these routines for the basic numerical types which include integer, real, double precision and complex have led many engineering application developers to ignore the deficiency in the language in other basic constructs. Fortran has

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

only one basic construct for selection: the IF-THEN-ELSE statement. The statement for selection from many cases, so called CASE or SWITCH construct, is missing and it has to be simulated with a series of IF-THEN-ELSE statements. Fortran only has one basic construct for repetition: DO-loops; all repetitions have to be put into this form in Fortran.

Fortran was a pioneering language but a poorly designed one by today's standard. Although many remnants of the original designs are in Fortran-77, no criticism should be directed to the original designers for they had enough trouble designing one of the first high level languages that could be implemented. No one at the time could be expected to anticipate the requirements of a good language applicable some 35 years later. In fact many software concepts and theories of programming languages were developed after Fortran.

The ANSI (American National Standards Institute) Standards Subcommittee X3J3 has proposed a new Fortran standard informally known as Fortran-8x in 1987 [Global 87]. This subcommittee is under the administrative control of ANSI X3, the group responsible for computer systems. Major new features added are operations on entire or portions of arrays, dynamic storage, and supports for modules and user-defined types. Fortran-8x allows more flexibility in the documentation of source code: the source code may be in free-form, the comments may start anywhere in a line, and variable names may be up to 31 characters. User-defined types and modules are designed to promote the creation of reusable software and these features have led some to compare Fortran-8x with Ada. Fortran-8x is compatible with Fortran-77 which means that Fortran-77 programs will conform to Fortran-8x. Partly because of this, a major criticism of Fortran-8x is that the language has become too big. Others have noted that instead of standardizing existing practices, which X3J3 is supposed to do, the committee is designing a new language.

Major features missing in Fortran-8x are pointer types, bit data type and bit manipulation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operations, other control constructs, block structure, and variant data structure. These features are all supported in C. In addition, the deficiencies of Fortran discussed in this section do no exist in C which is partly why C is a modern procedural language.

### 3.3.2. C, A Modern Procedural Language

C was originally designed for the UNIX operating system on the DEC PDP-11 by Dennis Ritchie at AT&T Bell Laboratories. The first implementation was operational in 1972 and after almost 20 years its popularity is still attributed to the merits in the language [Holtz 88]. C is well known as a systems programming language because operating systems such as UNIX and many compilers are written in C. But C is a general purpose language that has been effective in many other applications such as numerical and database programs. C is not tied to any particular machine and many powerful portable programs are written in C. However, C also allows programs that are intentionally not portable. Today, C compilers are available for practically all computers from the smallest personal computers to the very large mainframes.

Key characteristics of C are discussed next and the features that are different from Fortran are noted. C, like Fortran, is a procedural language. The idea behind the procedural paradigm in conquering a complex problem is to divide the problem into many simple procedures. All procedures in C are functions. The function called *main* is different because the program starts there. Functions are developed in C to resolve complexity, for reusability and also for clear coding. Programming in C involves writing many functions. C supports recursion and the best way to write a recursive function is to assume that the function already exists and freely call the function from inside the body that is being written. Recursive functions often provide elegant codes and the only caution is that the recursion must terminate for all cases. The parameter passing in C is call-by-value and to have the side effect of call-by-references, pointers must be used. Pointers to functions as arguments

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can also be used to simulate the passing of functions as arguments. Pointers are closely linked to arrays and they can be used for more efficient computations of array subscripts. This is important in engineering application programs where there are many arrays of numbers and programs spend a good deal of time computing subscript values.

C is a small language; there are 32 key words, 41 operators (considered as a very rich set), and the complete syntax summary can be fit on five pages. Most books on C describe the language as versatile and expressive. This is because C provides a rich set of basic elements of the language and few simple rules to derive unlimited possibilities. The basic types are characters, up to three sizes of integers, and up to three sizes of floating point numbers. The integers can be qualified as unsigned and others can be qualified as constants. C gives attention to the structuring of data. There are four simple ways to derive new data types: *arrays*, *pointers*, *unions*, and *structures*. Unions hold different data types in the same memory and structures are collections of one or more data types. C allows structures to be assigned as a unit, passed to functions as arguments and returned by functions. Structures are called records in other languages but they are generally more restrictive in other languages. If a definition of a derived data type becomes too elaborate *typedef* can be used to give it a more mnemonic name. The possibilities of derived data types are infinite and each kind has a different utility. In C, there is no separate `logical` type like Fortran. Integer values that are not `0` are *true* and `0` is *false*.

C provides all the basic control-flow constructs. For branching, there are the *if-else* statement for branching into `true` or `false` and the *switch* statement for branching into one of multiple cases. There are three loop constructs: the *do* statement where the termination test is after the body of the loop, the *while* statement where the test is before, and the *for* statement where the test is before and the control parameter jumps in value after each loop. C is not a block structured language like PL/I or ALGOL mainly because functions cannot be defined inside a function. But C supports block structures ( `{ ... }` ) to group

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

statements together and local variables may be declared inside the block. This block structure may be placed anywhere a single statement may appear. The control and block structures in C support structured programming.

The assignment in C ( = ) is a binary operator and like other operators it yields a value when used. In C, a value does not have to be used in the program. Function yields a value but it does not have to be used. In a place where a value can be placed, an expression is allowed. Expression is also a statement. This is why in C, it is just `name_of_function()` to invoke the function and not *call* `name_of_function()` like Fortran. This is also why in C, an expression like `y=z=4` is legal and it assigns the value `4` to `z` and the value of `z=4`, which is 4, to y. A single C statement,

```
while (*B++ = *A++);
```

is an instruction to traverse all the characters in string `A` and copy them to string `B`. Because of such statements some people have criticized C codes as cryptic - others have praised it as expressive and versatile.

The language described in *The C Programming Language*, the original edition published in 1978 [Kernighan 78], has for years been the widely accepted standard for C. In 1983, ANSI established a committee, X3J11, to begin the formal standardization of C. The process of defining the standard C proceeded with many publications by the ANSI committee: *Preliminary Draft Proposed Standard - The C Language* [ANSI 85] was available to the public as an information bulletin in 1985, *Draft Proposed American National Standard for Information system - Programming Language C* (document No. X3J11/88-001) was released on 11 January, 1988, and after a few more drafts, *American National Standard for Information Systems - Programming Language C, X3.159-1989* was submitted to the ANSI Board of Standards Review for final approval on 31 October, 1988. As of early 1990, it has not been approved and a formal ANSI standard for C may or may not exist by the end of the year.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A current stumbling block is mainly political such as unhappiness expressed by foreign countries through ISO (International Standards Organization) [Plauger 90]. The delay in the standardization has not hindered the vendors who have marketed C compilers as ANSI C, with the word *draft* written somewhere inconspicuously. Fortunately, the proposed ANSI C leaves the original C language remarkably intact with only few exceptions. X3J11 began with the widely accepted definition of C given in Appendix A of [Kernighan 78] and the final standard is expected to be not too different from the original C. The details of the proposed ANSI C and the difference between it and the original C may be found in *The C Programming Language*, the second edition published in 1988 [Kernighan 88],

The proposed ANSI C language adds few new keywords, formalizes some of the ideas suggested in the original book, and clarifies some of the ambiguities that existed for some time. A new keyword *void* is added to denote generic type. Void is to be used primarily as *void** to denote generic pointer. Wider capability of structures are now possible as they may be assigned, passed to functions and returned by functions. Scopes of formal parameters inside a function's body and *extern* declarations in an inner block are clarified. Probably the most significant change from the programmer's point of view is the introduction of function prototype in declaration. In the original C, only function name and return type are declared. In the proposed ANSI C, the types of arguments must also be declared in proper order. The required syntax is borrowed from the C++ language. This remedies a long standing flaw in C where types of arguments across functions are generally not checked by compilers. Another new keyword *const* used to qualify types as constants is also borrowed from C++. BCPL (British Computer Programming Language) and B (the language used to implement the first UNIX in 1970) have greatly influenced the design of the original C. BCPL and B are typeless languages - the only type being the machine word. Fortran and C are typed languages in a sense that there are many types in the languages and a variable has to be declared before it can be used. In Fortran-77, implicit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

declarations ( variables starting with I - N are integers and the rest are real ) are used quite extensively. The original C was not a strongly typed language in a sense that conversion of types, especially between pointers and integers, was quite permissive and compilers did not check the argument types in function invocations. The proposed ANSI C moves towards more strongly typed language with the prototyping of functions and the introduction of *void* type.

The proposed ANSI standard also prescribes a set of constants that characterize the machine on which a C program is run so that more powerful portable programs may be developed. The major addition in the proposed ANSI standard is the C's run time library, which is not part of the language. There are fifteen groups in the proposed ANSI standard C library of which the Input/Output ( <stdio.h> ) group is nearly one third and it has the most heavily used functions. The functions used for dynamic storage are also included in the library. Important to engineering applications is the math group ( <math.h> ). There are 22 functions in this group and the functions provided are similar to the built-in math functions in Fortran-77. However, Fortran-77 has many more functions because in addition to generic functions (functions that accept any appropriate argument type) there are different functions for different argument types. For example, there are three square root functions in Fortran-77: SQRT(real*4) for single precision real and also generic, DSQRT(real*8) for double precision real and CSQRT(complex*8) for single precision complex. There is only one square root function in <math.h>, sqrt(x), which always returns *double* ( which is equivalent to real*8 in Fortran-77 ) and the argument x is first converted to *double* if it is *int* ( integer ) or *float* ( which is equivalent to real*4 in Fortran-77). The C math functions do not support complex numbers.

Besides the merits in the language, engineering application developers have recently programmed in C because there are many C functions that allow the programmer to conveniently interact with the operating system, graphics packages, and database systems. In

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

addition, Fortran subroutines can be called from C programs. In most UNIX systems, a Fortran subroutine can be called from a C program by adding an underscore ( _ ) after the name of the Fortran subroutine and using pointer to argument as the actual argument if the argument is not an array.

### 3.3.3. The C++ Language

C++ was designed by Bjarne Stroustrup in the early 1980's at AT&T Bell Laboratories, the place that has retained most individuals who have been influential in defining and evolving the C language. Outside of Stroustrup's language group, C++ was available at AT&T in 1983 and to the public in 1985. *The C++ Programming Language* [Stroustrup 86], published a few months after the release of C++ has served as an informal reference for the language. Other books on C++ soon followed [Berry 88; Weiner 88; Lippman 89; Smith 90].

C++ "compilers" can be divided into three groups:

1. Translators that produce C as the target code, and use C compilers to produce object code.
2. True C++ compilers that directly produce object code.
3. Sold as C++ compilers for marketing purposes, but they are actually translators that hide the translation via C characteristic.

The initial implementation of C++ was with translators. These translators are also called a compiler front-end or a pre-processor to the C compiler. Widespread acceptance of object oriented programming has been delayed because object oriented languages are somewhat taxing on CPU. A recent trend in C++ is towards more efficient compilers that can produce faster run time codes, and among many object oriented languages that have emerged since the *hype* about object oriented programming in the 1980's, C++ has

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

become the most run time efficient language.

It is hard to measure the popularity of a language but there are indications that the C++ user community is continuously growing while interests in many other object oriented languages have somewhat stabilized. At AT&T, C++ is becoming the de facto standard language for development of new features and products. The C++ Conference held under the auspices of the USENIX Association attracted approximately 200 participants in 1987 when it was held in Santa Fe; the conference held in Denver in 1988 attracted 500 [USENIX 88]. Recent issues of computer magazines such as *Byte* and *Computer Languages* carry several advertisements for C++ compilers by Zortech, Borland, ImageSoft, etc., indicating growing popularity of C++ among personal computer users as well.

C++ was originally designed to support large scale event-driven simulation projects. Machine efficiency and compatibility with C were high on the priority list. With minimum exceptions, newer versions of C++ ( AT&T V2.0 was released in 1989 ) also kept C as a subset. C++ is an evolving language and many suggestions for new features can be found in the journal: *The C++ Report*. Multiple inheritance was formally implemented in version AT&T V2.0. Parameterized types are being proposed [Stroustrup 88b] and some laboratory versions may already have it. Parameterized types allow definition of classes that can have the name of type as a parameter; for example, an array of type can be defined and a user can specify the type of elements in the array.

Because C++ keeps C as a subset, it is a procedural language. There are additions that are designed to enhance this procedural paradigm. Function prototyping ( see Section 3.3.2 ) in the proposed ANSI C came from C++. Overloading of function names is allowed. Trailing arguments of a function may be assigned default values. This is useful in maintenance because a function can be modified to have some more arguments and other programs that call the original function do not have to be changed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C++'s early name was *C with classes*. A major enhancement in C++ is the *class* construct, and an object oriented paradigm is supported with it. The *struct* in C is now a special case of class where all data are public. Class in C++ is the mechanism to define user-defined types. Overloading for most of the C operators are allowed for operators defined for class. Single and multiple inheritances can be used to derive a class from an existing class. Most methods defined for classes are bound at compile time for efficiency considerations. Only methods that are declared *virtual* in the base class and defined in the derived class are generally bound at execution time. An object oriented paradigm necessitates extensive function calls. In C++, a function can be declared *inline* to reduce the function call overhead. Inline functions have the semantics of functions but they are expanded on location by the compiler so the function call overhead is eliminated. C++ is a full fledged object oriented language by any standard.

It is useful to have many features in a language, but diversity of features does not make a language useful or powerful. Added features are often causes of errors and confusion. However, the added features in C++ seem to blend well with C and reports on the short experiences with the language are mostly encouraging. Criteria for a good language are usually very general or else we wouldn't have so many languages whose users claim to be powerful. C++ was designed for simulation projects but many have found it useful in other applications such as systems programming at AT&T and graphics packages. Widespread use in other areas means that helpful tools will emerge. Practically, C++ is compatible with C and C is compatible with Fortran. So C and Fortran codes that exist can be utilized. Effective use of the object oriented paradigm requires useful class libraries. Because such libraries are lacking and experience in C++ in engineering applications is short, conclusion about the appropriateness of C++ for engineering programs cannot be made; however, of all the existing languages C++ seems to be the best choice for many kinds of engineering application systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following list summarizes the merits in the C++ language that are considered appropriate for engineering applications:

1. C++ supports the object oriented paradigm.
2. C++ supports a modern procedural paradigm.
3. Diverse features in C++ are blended together to give the *look and feel* of a uniform language.
4. Among the object oriented languages, C++ produces one of the most efficient execution time code.

Merits in a language is important, but just as important are the practical concerns. The following list gives practical indications that C++ can become the language to write the next generation of engineering systems:

1. C++ can utilize many existing engineering modules which are written in Fortran or C.
2. C++ was designed so that the conventional edit - compile - debug cycle is used for program development. This and similarities with C make it easier for the seasoned engineering programmers to learn the language.
3. An ANSI subcommittee was formed to standardize the C++ language. When the language is standardized, standard C++ programs will be more portable.
4. Wide acceptance in other areas of application have produced efficient and economic implementations on many hardware.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 4

# OBJECT ORIENTED DEVELOPMENT OF FINITE ELEMENT PROGRAMS

## 4.1. Introduction

Experience in applications of object oriented programming in engineering is still limited. Fenves shows object oriented programming for engineering software development using Smalltalk in reference [Fenves 90]. This chapter provides general guidelines for object oriented development of engineering software using the C++ language. Object oriented development of software is based on modules that correspond to the objects within a model for a physical system [Booch 86]. This differs from the conventional development of software in which modules correspond to important functions of a real-world system.

A finite element program developed as a teaching tool using object oriented concepts and the C++ language is used as an example. The chapter specifically discusses object oriented development of finite element programs but the ideas are applicable to other computer based methods and programs in engineering.

The finite element method is one of the most widely used methods of engineering analysis. Most finite element programs are written in Fortran and engineering programmers who have dealt with these programs at the source code level find these codes brittle, i.e., they are difficult to add a new feature, hard to modify an existing routine, and often very difficult to maintain. However, there are enormous amounts of these routines that work and tens of thousands of engineers who are familiar with how these programs are developed.

A hybrid object oriented language such as C++ can fully utilize existing codes that are written in Fortran or C. The procedural paradigm, with which most engineering



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

programmers are comfortable, is important in many engineering applications because the algorithmic nature of many solutions are best translated into a program using a procedural paradigm. The problem with many Fortran codes is maintenance, not their lack of ability to solve a given engineering problem. After all, an object oriented paradigm does not offer any new solution techniques, it only *promises* to cut the development and maintenance costs of large systems by producing codes that are reusable and ones in which the interactions among modules are manageable.

Long before the object oriented programming was introduced, the importance of clarity and modularity in a program was recognized. Variations on how to produce such codes had diverse schools of thought that emphasized structured programming, modular design, top-down programming, etc. A general programming method introduced as good for all programming activities was often too general to provide any guidelines. Algol and PL/I are thought to have failed because each tried to be the language for all programming activities. Although C++ has the merits to become the next language in engineering, whether it will become one probably depends more on practical factors discussed in Section 3.3.3. This chapter advocates certain guidelines for writing C++ programs for engineering applications that will produce clear, modular, and structured programs using both the procedural and object oriented paradigms supported by the C++ language.

The rest of this chapter is organized as follows. Section 4.2 presents a brief overview of the finite element method; a finite element displacement formulation for elasticity problems is outlined in Section 4.2.1 and how the finite element method is presently taught is discussed in Section 4.2.2. Section 4.3 presents guidelines for object oriented development of finite element programs in C++; general guidelines are given in Section 4.3.1 and a C++ object oriented finite element program, which was developed to be used as a teaching tool, is described in Section 4.3.2 as an example. The general guidelines for object oriented development of engineering software systems are based on the experience from the actual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

design and coding of the object oriented finite element program.

### 4.2. The Finite Element Method

The finite element method initially proposed in mathematics by Courant [Courant 43] and in engineering by Turner et al. [Turner, 1956] has proved to be quite efficient as a computer based analysis technique for the solution of many engineering problems specified by sets of partial differential equations. The extensive publications on the applications and the analyses of the method, decades after its initial conception, have shown that the method has infinite variety but has specific rules and theorems governing its use. Many areas such as finite elements with Lagrange multipliers, finite deformation elements, materially nonlinear elements and shells are still subjects of intense research [Becker 86; Hughes 87; Zienkiewicz 89].

The finite element method, however, has to be put into a program and the programming techniques for the finite element method nearly have remained unchanged. Using a conventional programming language, such as Fortran, many finite element programs are complex and difficult to maintain. This is due in part to the requirement that the program must specify all of the details of the exact computation sequences and data structures. These low-level specifications are intertwined with the high-level mathematical formulation of the method which usually is a few pages of matrix algebra, integrals and derivatives [Rehak 89].

#### 4.2.1. A Finite Element Displacement Formulation

The most common formulation of the finite element method is based on displacements as primary global variables. General steps in a finite element displacement formulation of a linear elasticity problem are outlined below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Input the data defining geometry, elements, boundary conditions and loads.

2. Assemble the element stiffnesses $\mathbf{k}_e$ into the global stiffness $\mathbf{K}$,

$$\mathbf{K} = \sum \mathbf{k}_e$$

where the $\mathbf{k}_e$ are obtained by the following procedure :

2.1. Express the unknown displacements $\mathbf{u}$ in terms of the matrix of shape functions $\mathbf{N}$ and the nodal variables $\mathbf{a}$:

$$\mathbf{u} = \mathbf{N} * \mathbf{a}$$

2.2. Express the strains $\epsilon$ in terms of the strain matrix $\mathbf{B}$ and $\mathbf{a}$:

$$\epsilon = \mathbf{B} * \mathbf{a}$$

2.3. Express the stresses $\sigma$ in terms the elasticity matrix $\mathbf{D}$ and $\epsilon$:

$$\sigma = \mathbf{D} * \epsilon$$

2.4. Express the element stiffness $\mathbf{k}_e$ as an integral of matrix product:

$$\mathbf{k}_e = \int_{\Omega_e} \mathbf{B}^T * \mathbf{D} * \mathbf{B}\, dV$$

3. Form the load vector $\mathbf{F}$ and apply boundary conditions.

4. Solve $\mathbf{Ka} = \mathbf{F}$.

5. Output the requested results.

Formulation procedures for other problems follow similar steps. The steps 1 through 5 may be considered as a top level abstraction of the finite element method. At this level, the objects $\mathbf{K}$, $\mathbf{k}_e$, $\mathbf{F}$ and $\mathbf{a}$ are visible. The steps 2.1 through 2.4 are an intermediate level of abstraction below Step 2. The objects $\mathbf{u}$, $\epsilon$, $\sigma$, $\mathbf{N}$, $\mathbf{B}$, $\mathbf{D}$, $\mathbf{a}$, and $\mathbf{k}_e$ are visible at this level. The top level description does not require any information on steps 2.1 through 2.4 but choices made in these steps produce many different elements, which are reflected in $\mathbf{k}_e$.

The above formulation, although quite general, shows much of the essence and the elegance of the finite element method. Completely missing from this description is the data structures of the matrices and the detailed instruction sets for the operations, say the * in $\mathbf{N}*\mathbf{a}$ (see Step 2.1). The symbols $\mathbf{N}$, $\mathbf{B}$, and $\mathbf{D}$ are concepts whose functions are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clearly defined; what they represent are known except how they are implemented is a detail hidden at this level. The above description of the finite element method shows that the method is clearly described without any low level instructions. However, the instructions made up of low level abstractions provided by a programming language, i.e., the basic constructs of the language, make up almost the entire code in current finite element programs.

### 4.2.2. Current State of Teaching the Finite Element Method

One can use a finite element program without understanding the details of the method, but here lies the danger of misinterpreting the results from the program. The past three decades have produced enthusiasm in researchers working on the understanding and the development of the finite element method; as a result, the method has gained scientific backing and a wide usage in diverse areas. However, less attention has been given to effective teaching of the total method to a wide audience of engineers from diverse backgrounds. Appropriate use of finite element programs requires understanding of the basics of the method. In addition, educators must also enable students to write correct finite element programs.

At a university level, an instructor attempts to stimulate creativity and innovations both in the definition of new problems classes and in the search for their solutions [Taylor 87]. To achieve this when the subject is the finite element method is a challenging task partly because the subject matter requires background knowledge from diverse areas of engineering, which include:

1. Physics of the field (structural engineering, fluid mechanics, heat transfer, etc.);
2. Numerical methods (equation solving, numerical integration, numerical interpolation, etc.); and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Computer fundamentals (programming principles, a programming language, an operating system, etc.).

An instructor cannot assume that all students have the necessary background, but all the materials cannot be taught in a finite element course either. Because of this, the finite element method is still taught mainly at a graduate level in most engineering departments despite the need to reach a wider audience.

The basic theories behind the finite element method can be taught using a textbook or class notes. However, the finite element method is a computer based method of analysis and many aspects need demonstration on a computer to convince a student that they can implement the theory into working programs. Thus, essential parts of a finite element course are studying finite element programs and actual coding. However, there does not seem to be an effective way of including this into a finite element course. Some instructors use assignments that require running sample problems on well known finite element codes such as ANSYS [Kohnke 1979] and NASTRAN [Mclean 81; NASA 79]. These assignments, however, are ineffective because one can do them without understanding the method; one only needs to understand how to prepare the input data which can be learned by reading the program manuals. Other instructors who acknowledge this problem design assignments where the student must code a portion that is to be integrated with the rest of a finite element program. The portion generally is a subroutine or a segment in a subroutine where the theory is covered in the class. This type of assignment seems more challenging than running sample problems but it is still not totally effective. Most finite element codes are written in Fortran where the program consists of many subroutines and the complex network of information is shared among subroutines through common arrays and subroutine arguments. In order to write a portion of a finite element code, the student still has to know quite a bit about the rest of the program. This requires weaving through the arrays, the arguments, and other parts of the code which the student may not be ready to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

comprehend.

In structural engineering, SAP (Structural Analysis Program) [Wilson 70], STRUDL (STRUctural Design Language) [Logcher 71], and FEAP (Finite Element Analysis Program) [Taylor 77] were started partly as instructional tools and have been widely used in educational environments for many year. McAUTO-STRUDL and GT-STRUDL, which are derivatives of the original STRUDL, have become successful commercial products. The latest version of SAP, SAP-90 [Wilson 90], is also a viable commercial product where many recent numerical methods have been incorporated. FEAP is used widely in research institutions around the world and many modern elements and nonlinear algorithms have been first tested in this program. However, these programs as teaching tools at the source code level do not address the issues discussed above. In general, academia *in engineering* emphasize efficiency of calculations and do not stress factors which software professionals consider important (such as, user convenience, cost of development and maintenance).

Abstracting relevant concepts from the whole picture of the finite element method is important for effective instruction. However, this is extremely difficult with a conventional procedural program. Object oriented programming with its powerful means of abstraction can dramatically improve finite element codes in this respect. Examples are shown in Section 4.3.2.

### 4.3. Object Oriented Development of Finite Element Programs

The main purpose of programming paradigms and style is to produce clear, modular and structured programs so that development and maintenance of these codes are efficient. The guidelines provided in this section are to fulfill this purpose for C++ engineering programs and they should be considered as *additions* to existing good styles of programming that are necessary to develop today's much larger engineering systems ( see Section 3.2 ).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

#### 4.3.1. General Guidelines for Object Oriented Development

Support for an object oriented paradigm in C++ offers elegant abstraction and code reuse techniques that are useful in large engineering projects.

Abstraction is the essence of all computer programming. From a machine language to an assembly language, and from an assembly language to a general purpose language, higher levels of abstraction allow application programmers to write programs that are comprehensible. Since an application programmer does not deal with the machine or the assembly language, the lowest level of abstraction is the basic constructs in the programming language. For decades, engineers produced codes that are based only on this level. Complexity increases as a system becomes larger and for these systems higher levels of abstraction that correspond to the familiar concepts in the application domain must be reflected in the source code.

Today's engineering programs become modules in a large system where they have to interact with other resources in a changing environment. A program's interface with other resources must be based on general conceptual entities in the application domain where the specific formats of input and output data must be flexible, i.e., easily modifiable.

Programmers no longer build an entire program with the basic constructs in the language. Instead, components from previous projects and modules written by other programmers are reused. Reuse of code reduces development costs by exploiting the existing resources. In a C++ object oriented paradigm, the source of reusable code is the method definitions of classes that are in *compiled* form. This provides effective means of maintenance because codes that are tested (existing codes) are *not* altered and the system can be maintained primarily by *additional* coding.

A guideline for object oriented development of C++ engineering software based on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Levels of abstraction
2. Reusability of classes

is proposed.

**4.3.1.1. Levels of Abstraction**

Object oriented concepts are built upon the ideas developed for structured programming. In a structured program, effectively organizing an application's functions is important where these functions are transformed to procedural codes. In an object oriented program, an application must be organized with objects, where functions and data are encapsulated.

In engineering, complexity is often simplified by perceiving the problem at various levels of abstraction. Analysis of a project entails identifying these levels. At each level there are classes (concepts) and objects with the right amount of detail that are used to arrive at a part of a solution. An object oriented engineering program must be designed so that these levels of abstraction are clearly identified in the source code. This is important for clarity of the program and in this respect, operator overloading plays a major role. As an independent module, the program must be versatile where the interface with other resources in the computer system can be established at any level of abstraction.

Defining classes is a major part of coding in C++. When there is a concrete concept, an invariant that is not short-lived, make it a class. When there is an identifiable entity, a project specific element, make it an object of some class. If two classes have something significantly in common, make that a base class; it will become more reusable in the future. When designing classes, do not use global data, global functions, or public data; this defeats the encapsulation and data hiding purposes.

An outline of the procedure for an object oriented development of C++ engineering software based on levels of abstraction is given below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Identify the levels of abstraction in the project ( Top level, Intermediate level, etc.).
2. Abstract general concepts that will become C++ classes for each level defined in Step 1.
3. Determine the class methods that are used as part of the interface for the entire program and the class methods that are for interaction among objects inside the program.
4. For the classes identified in Step 2, see if there are any classes in the class library that are similar. Reuse the similar classes (see Section 4.3.1.2.).
5. Define and code new classes.
6. Code relevant algorithms for each level of abstraction identified in Step 1 using objects that are instances of classes defined in steps 4 and 5.

As discussed in Section 3.2.1, the five stages of software development are: 1. Specify requirements, 2. Design, 3. Coding, 4. Testing, and 5. Delivery. In the above procedure, steps 1, 2 and 3 correspond to the design stage and steps 4, 5 and 6 constitute the coding stage. If proven and familiar algorithms are used, only class definitions need to be thoroughly tested.

#### 4.3.1.2. Class Library and Reusability

C++ allows one to clearly organize programs. By designing classes with well-defined interfaces the risk of system wide bugs are minimized. The declarations of classes, which are the user interfaces for the classes, should be placed in header files. The method definitions, which are the implementation details of the classes, should be compiled and placed in files with restricted access.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A C++ programmer must build a library of classes. Most engineering programmers already have useful subroutines and functions that are written in C or Fortran. These codes, which can become definitions of methods without modification, should be the starting point of building an engineering C++ class library. Pure object oriented programming supporters advocate forming a single inheritance hierarchy of all the classes in the library; this is not necessary. Organizing classes into hierarchical categories similar to the UNIX file system, however, is useful for access purposes.

General C++ classes are included with compilers [Zortech 89] and some disciplines have public-domain C++ routines. For example, NIH (National Institute of Health) provides health related C++ routines available to the public [Ladd 90]. Class libraries related to engineering are most likely to be developed as an individual's or an organization's proprietary code.

A Class library is the source of reusable codes in C++. Wherever possible, these codes must be used to avoid *re-invention*. Tested code enters the class library and must be used without modification. When developing new classes for a project, reusability must be emphasized. Classes archived in the library should not be project-specific, but be abstractions of invariant concepts in the application domain of the project. Project-specific classes should be derived from the classes in the library.

A class library is much more versatile and powerful than the function libraries of C or Fortran. Like library functions, classes in the library must be used without modification. However, these classes can be the bases of derived classes that accommodate a wide range of specific requirements of different projects and changing user requirements during the maintenance of the projects. Using inheritance and polymorphism, the additional coding for derived classes can be minimized.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The four types of inheritance relationships (see Chapter 2, Section 2.3.1.1.) may be coded as follows in C++ ( the small letter words are C++ key words, the capital letter words are class names, and text after // are comments):

1. Specialization ( a CONCRETE_BEAM is a kind of BEAM ):

```
class BEAM  {      ...     //BEAM    properties              };
class CONCRETE_BEAM: public BEAM
   {               ...     //special CONCRETE properties     };
```

2. Modification ( a BLUE_BEAM is like a RED_BEAM except the color ):

```
class RED_BEAM { ... public: void color(); //return RED };
class BLUE_BEAM:     public RED_BEAM
   {                 public: void color(); //return BLUE};
```

3. Extension ( a BEAM_COLUMN has a BEAM's and other additional features ):

```
class BEAM  {      ...     //BEAM    properties              };
class BEAM_COLUMN:   public BEAM
   {               ...     //additional COLUMN properties    };
```

4. Aggregation ( a FLOOR is made up of BEAMs, GIRDERs and SLABs );

```
class BEAM  {      ...     //BEAM    properties              };
class GIRDER{      ...     //GIRDER  properties              };
class SLAB  {      ...     //SLAB    properties              };
class FLOOR
   { BEAM     BEAM_object;
     GIRDER GIRDER_object;
     SLAB     SLAB_object;
                   ...     //other FLOOR properties          };
```

Multiple inheritance is supported in C++; if a BEAM_COLUMN is a BEAM and also a COLUMN, this can be coded as,

```
class BEAM                                    {   ...   };
class COLUMN                                  {   ...   };
class BEAM_COLUMN: public BEAM, public COLUMN {   ...   };
```

Virtual methods provide the means of polymorphism in C++. When a method is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

declared *virtual* in a class, definitions must be provided only by the derived classes. However, the declaration of a virtual method in the base class has enough information - the method name and the types of arguments - to uniformly treat various objects from different derived classes. Virtual methods are bound at run time in C++ and if run time efficiency is critical, these methods should be avoided.

Each derived class provides a different definition of the virtual method declared in the base class. If there are common features to be shared by all the definitions, then a separate method should be defined in the base class that contains these features. This method should be invoked by each definition of the virtual method. The following code shows an example:

```
class SHAPE {                 ... public: ...
    virtual void draw(); //draw() declared virtual
    void   SHAPE_draw(); //common features of draw()  };
class CIRCLE: public SHAPE { ... public: ...
            void draw();                               };
void CIRCLE::draw() { ... //CIRCLE's definition of draw
           SHAPE_draw(); //common features of draw()  };
```

There are still unresolved ambiguities in the scope of names in a derived class when multiple inheritance, global variables, and virtual methods are intricately intertwined and the programmer is advised to be cautious in choosing names in such situations.

In C++, the class definition of an object cannot be changed after the object is initially constructed; that is, given an object, there can only be one class definition throughout its existence. Also, a class is not an object in C++. These are the limitations of the object oriented programming model adopted for C++ and they are due to execution time efficiency considerations. However, many features offered by more pure object oriented programming models that allow an object's class to change are not that important in engineering program development. Such features are an object's capability to receive or relinquish

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instance variables or methods. Other features can be simulated with inheritance mechanisms in C++; for example, partial inheritance can be simulated with a modification type of inheritance if the part to be inherited is known at compile time.

### 4.3.2. An Object Oriented Program for Finite Element Instruction

OPFI (Object oriented Program for Finite element Instruction) is an object oriented C++ program for linear finite element analysis using isoparametric elements. The program is intended for instruction on finite element programming and on behavior of finite elements. OPFI is a general purpose finite element program where the user specifies element characteristics - B matrix, D matrix, Shape functions, and Gauss points and weights for numerical integration - in addition to other finite element input data. The element characteristics are conceptual entities that are developed as classes and data that determine the specific characteristics of the objects are input from the users. In conventional finite element programs, a user selects elements from the element library supported by the program; the user can input the parameters defined by the element but cannot alter any characteristic (B matrix, D matrix, shape functions, etc.) of the element.

The ability where a student can input the data that define element characteristics allows the student to learn the behavior of elements effectively. The source code of OPFI reflects many familiar algorithms used in the finite element method at various levels of abstraction and this can assist students to learn more quickly the details of finite element code development.

The rest of this section shows the object oriented development of OPFI and outlines how OPFI can be extended for dynamic analysis. The guideline follows the steps outlined in Section 4.3.1.1 and the extensibility is described in terms of reusability of classes that is elaborated in Section 4.3.1.2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The declarations of the classes used in OPFI are included in the Appendix. The detailed workings of OPFI are not essential in describing the object oriented development of the program and are not included in this dissertation. However, the users manual, sample problems in elasticity, and a complete listing of the program OPFI may be found in reference [Yoon 89].

#### 4.3.2.1. Levels of Abstraction in a Finite Element Displacement Formulation

The general procedure outlined in Section 4.3.1.1 for an object oriented development of engineering software is adhered to in the following discussion.

*1. Identify the levels of abstraction.*

The finite element displacement formulation described in Section 4.2.1 identifies two levels of abstraction:

1. Top level: Steps 1 through 5 (see Section 4.2.1).
2. Intermediate level: Steps 2.1 through 2.4 (See Section 4.2.1).

*2. Abstract general concepts that will become classes.*

The classes that represent concepts in the top level are described below (Italic words are class names and capital letters in parentheses are object names; see the Appendix for the declarations of the classes):

1. *input* ( I ): Reads the input data. The object in this class reads the conventional finite element data such as joint coordinates and element incidences.
2. *gstif* ( K ): Generates the global stiffness matrix, **K**.
3. *estif* ( E ): Generates element stiffness matrices, $\mathbf{k}_e$.
4. *output* ( O ): Outputs the requested results

The classes that represent concepts in the intermediate level are described below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. *Gpoint* ( G ): Generates the data for the Gaussian quadrature.
2. *Dmat* ( D ): Reads entries of the **D** matrix and computes numeric values.
3. *Bmat* ( B ): Reads entries of the **B** matrix and computes numeric values.
4. *Shape* ( Shp ): Reads the nodal shape functions.
5. *Stress* ( S ): Computes stress matrices from **D*B.**
6. *Stif_mat* ( eK, gK ): Stores element stiffness matrix which is formed from a numerical integration of $\mathbf{B}^T*\mathbf{D}*\mathbf{B}$ over the element domain, $\Omega_e$.

The objects in the classes *Gpoint*, *Dmat*, *Bmat* and *Shape* require input data from the user that determine the characteristics of the element used in the finite element analysis.

*3. Determine the objects that are used as part of the program's interface.*

At the top level, the objects I and O are used for the program's input/output relation. At the intermediate level, the objects G, D, B, and Shp are used for the program's input/output relation.

*4. Reuse of classes*

OPFI was developed from scratch so initially the class library was empty. The existing code used in OPFI is a Fortran equation solver. Many Fortran equation solvers are efficient and well tested. These routines should be used in finite element programs and others that require solving systems of equations. Integration of a Fortran subroutine into a C++ program is straight forward. In OPFI, the *solve* method of the class *gstif* (global stiffness) calls a Fortran equation solver after arranging the load vector and the coefficient matrix into a form that is consistent with what is required by the Fortran routine.

*5. Define and code new classes*

The definitions of classes used in OPFI may be found in [Yoon 89].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

*6. Code relevant algorithms for each level of abstraction.*

After the objects are instanced, the code that represents steps 1 - 5 of the top level of abstraction is shown below:

```
input I();                                            // Step 1: input
for (int i=1;i<=nel;i++) { E.form(i);                 // Step 2: direct stiffness
                           K = K + E; }
K.load();                                             // Step 3: form load vector
K.solve();                                            // Step 4: solve Ka = load
O.display();                                          // Step 5: display results
```

The codes for steps 1, 3, 4, and 5 are exact matches. Step 2 represents the direct stiffness method, a procedure well known to the professionals in this domain. The two lines of code that represent Step 2 concisely state the direct stiffness algorithm: form the stiffness for elment i { `E.form(i)` }, add the element stiffness to the the global stiffness { `K = K + E` }, and repeat for all the elements { `for (int i=1;i<=nel;i++)` }. The + in Step 2 is a good example of an overloaded operator that clarifies code.

Step 2.4 in the intermediate level of abstraction is the integration of $\mathbf{B}^T*\mathbf{D}*\mathbf{B}$ over $\Omega_e$. The code that represents this step is shown on the next page:

```
//  COMPUTATION OF eK = GAUSS INTEGRATION OF (B^t)*D*B
 for (i=1;i<=G.points();i++)              { // LOOP OVER GAUSS POINTS
   B.form(G,i)                           ;  // FORM B MATRIX
   Stress    S = (D*B)                   ;  // FORM S MATRIX
   S.store( )                            ;  // SAVE S MATRIX
   S = S * ( G.wt(i)*B.jacobian() );        // S=S*WEIGHT*JACOBIAN
   B.trans( )                            ;  // FORM B TRANSPOSED
   Stif_mat gK = (B*S)                   ;  // FORM gK=B^t*S
   eK = eK + gK                          ;} // ADD gk TO eK, END LOOP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The above code outlines the algorithm for a Gaussian numerical integration of $\mathbf{B}^T*\mathbf{D}*\mathbf{B}$ over the domain $\Omega_e$. This is a familiar algorithm and it is how integration is generally computed on computer. At this slightly lower level of abstraction, some detail in coding is present. Codes from even lower levels of abstraction, e. g., the code for multiplying **B** and **S** ( **B*S** ), contain the details of the data structures of the objects and basic constructs of the C++ language.

Object oriented programming techniques are versatile but their abuse can produce complex and cryptic programs. However, when the techniques are utilized with disciplined style and organization, the codes generated are modular and structured; the clarity of the codes is only limited by the imagination of the programmer.

**4.3.2.2. Extension for Dynamic Analysis.**

OPFI is a static finite element analysis program but many classes defined in OPFI can be reused as is or as base classes for a dynamic finite element analysis program. This is expected since static analysis is conceptually a part of dynamic analysis. Extending OPFI for dynamic analysis is the subject of this section. An example of levels of abstraction clearly reflected in the source code was shown with OPFI in Section 4.3.2.1. This section shows examples of the reusability of classes in an object oriented development of engineering software; actual coding is not included.

*The Equations of Dynamic Equilibrium :*

The equations of equilibrium governing the dynamic response of a system of finite elements can be expressed as

$$\mathbf{M}\,\ddot{\mathbf{u}} + \mathbf{K}\,\mathbf{u} = \mathbf{F}(t)$$

where **M** and **K** are the mass and stiffness matrices; **F** is the time dependent external load vector; and $\ddot{\mathbf{u}}$ and **u** are the acceleration and displacement vectors of the finite element

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

assemblage. For simplicity, the damping effect (commonly denoted with $\mathbf{C}\,\dot{\mathbf{u}}$) is assumed negligible.

*Solution Methods :*

The numerical solution methods for the response of structural systems subjected to dynamic loads can be grouped into two broad classes according to the solution space employed [Belytschko 83; Clough 75; Bathe 82]:

1. Time domain solution methods; and
2. Frequency domain solution methods.

The time domain solution methods operate directly on the equations of equilibrium, determining the response at discrete time steps $\Delta$, and may be used for linear and nonlinear problems. The frequency domain solution methods operate in a transformed domain, using Fourier Transforms, and are restricted to linear systems.

The time domain solution methods are also called step-by-step direct integration methods and some of the common time domain solution methods are the following : Newmark's method, Wilson's $\theta$ method, finite difference methods, Euler's method, and Runge-Kutta type methods. In structural engineering, Newmark's method has been found to yield good results with relatively little computational effort and this method is considered in the following presentation. Discussion on the merits of Newmark's and other methods are beyond the scope of this presentation; Newmark's method is chosen only because it is conceptually simple but sufficient to address the programming aspect of the general time domain solution methods of finite elment systems.

*Step-by-step Integration, Newmark's Method :*

The algorithm for Newmark's method in a predictor-corrector form is outlined below [Newmark 59; Belytschko 83]. The subscript denotes the value of time. The displacement,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

velocity and acceleration at time $T$ are denoted by $\mathbf{u}_T$, $\dot{\mathbf{u}}_T$ and $\ddot{\mathbf{u}}_T$ respectively.

1. The initial values of dispacement and velocity ($\mathbf{u}_0$ and $\dot{\mathbf{u}}_0$) are known and the initial value of acceleration ($\ddot{\mathbf{u}}_0$) is computed from the equilibrium

   1.1.
$$\mathbf{M}\,\ddot{\mathbf{u}}_0 = \mathbf{F}_0 - \mathbf{K}\,\mathbf{u}_0$$

2. The predictor formulas for the displacement and velocity ($\hat{\mathbf{u}}_{T+\Delta}$ and $\hat{\dot{\mathbf{u}}}_{T+\Delta}$) for time T=0, $\Delta$, 2 $\Delta$, 3 $\Delta$, ... are given by the following :

   2.1.
$$\hat{\mathbf{u}}_{T+\Delta} = \mathbf{u}_T + \Delta\,\dot{\mathbf{u}}_T + \left(\frac{1}{2} - \beta\right)\Delta^2\ \ddot{\mathbf{u}}_T$$

   2.2.
$$\hat{\dot{\mathbf{u}}}_{T+\Delta} = \dot{\mathbf{u}}_T + (1-\gamma)\,\Delta\ \ddot{\mathbf{u}}_T$$

3. The corrector formulas for the displacement, velocity and acceleration ($\mathbf{u}_{T+\Delta}$, $\dot{\mathbf{u}}_{T+\Delta}$ and $\ddot{\mathbf{u}}_{T+\Delta}$) for time T=0, $\Delta$, 2 $\Delta$, 3 $\Delta$, .... are given by the following:

   3.1.
$$\left(\mathbf{K} + \frac{1}{\beta\,\Delta^2}\mathbf{M}\right)\mathbf{u}_{T+\Delta} = \mathbf{F}_{T+\Delta} + \frac{1}{\beta\,\Delta^2}\mathbf{M}\,\hat{\mathbf{u}}_{T+\Delta}$$

   3.2.
$$\ddot{\mathbf{u}}_{T+\Delta} = \frac{1}{\beta\,\Delta^2}\left(\mathbf{u}_{T+\Delta} - \hat{\mathbf{u}}_{T+\Delta}\right)$$

   3.3.
$$\dot{\mathbf{u}}_{T+\Delta} = \hat{\dot{\mathbf{u}}}_{T+\Delta} + \frac{\gamma}{\beta\,\Delta}\left(\mathbf{u}_{T+\Delta} - \hat{\mathbf{u}}_{T+\Delta}\right)$$

4. Set the time $T = T + \Delta$ and go to step 2.

In Newmark's method, $\beta$ and $\gamma$ are the free parameters and the particular choices for these parameters correspond to different methods; for example, the choices $\beta = \frac{1}{6}$ and $\gamma = \frac{1}{2}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

correspond to the linear acceleration method, which is conditionally stable, and the choices $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$ correspond to the average constant acceleration method, which is unconditionally stable for linear problems.

*A Finite Element Formulation :*

A formulation for the linear dynamic response of a finite element system using Newmark's method with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$ is presented below. The steps are organized so that the similarities with and the differences from the formulation for the static analysis given in Section 4.2.1 for OPFI are accentuated (same notations are used).

1. Input the data defining geometry, elements, boundary conditions, initial conditions ($\mathbf{u}_0$ and $\dot{\mathbf{u}}_0$), time step ($\Delta$), and time dependent load ($\mathbf{F}_T$ for T=0, $\Delta$, 2 $\Delta$, 3 $\Delta$ ...).

2. Assemble the element stiffnesses $\mathbf{k}_e$ into the global stiffness $\mathbf{K}$,

$$\mathbf{K} = \sum \mathbf{k}_e$$

where

2.1

$$\mathbf{k}_e = \int_{\Omega_e} \mathbf{B}^T * \mathbf{D} * \mathbf{B} \, dV$$

3. Assemble the element masses $\mathbf{m}_e$ into the global mass $\mathbf{M}$,

$$\mathbf{M} = \sum \mathbf{m}_e$$

where

3.1

$$\mathbf{m}_e = \int_{\Omega_e} \rho \, \mathbf{N}^T * \mathbf{N} \, dV$$

4. Form the effective stiffness $\tilde{\mathbf{K}}$,

$$\tilde{\mathbf{K}} = \mathbf{K} + (\mathbf{M} * ( \frac{4}{\Delta^2} ))$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Factorize $\bar{\mathbf{K}}$ and $\mathbf{M}$ where the original forms of the matrices are not lost.

6. Obtain $\ddot{\mathbf{u}}_0$ by solving

$$\mathbf{M}\,\ddot{\mathbf{u}}_0 = \mathbf{F}_0 - (\,\mathbf{K} * \mathbf{u}_0\,)$$

7. Initialize the value of time $T = 0$.

8. For each time step, compute the displacement and velocity ( $\mathbf{u}_{T+\Delta}$ and $\dot{\mathbf{u}}_{T+\Delta}$ ) with the following steps:

   8.1 The values of $\mathbf{u}_T$, $\dot{\mathbf{u}}_T$, $\ddot{\mathbf{u}}_T$, and $\mathbf{F}_{T+\Delta}$ are known.

   8.2 Compute $\hat{\mathbf{u}}_{T+\Delta}$, the displacement predictor, given by

$$\hat{\mathbf{u}}_{T+\Delta} = \mathbf{u}_T + (\,\dot{\mathbf{u}}_T * \Delta) + (\,\ddot{\mathbf{u}}_T * (\,0.25 * (\,\Delta^2\,)\,)\,)$$

   8.3 Compute $\hat{\dot{\mathbf{u}}}_{T+\Delta}$, the velocity predictor, given by

$$\hat{\dot{\mathbf{u}}}_{T+\Delta} = \dot{\mathbf{u}}_T + (\,\ddot{\mathbf{u}}_T * (\,0.5 * \Delta\,)\,)$$

   8.4 Compute the corrected displacement, $\mathbf{u}_{T+\Delta}$, by solving

$$\bar{\mathbf{K}}\,\mathbf{u}_{T+\Delta} = \mathbf{F}_{T+\Delta} + (\,(\,\mathbf{M} * \hat{\mathbf{u}}_{T+\Delta}\,) * (\,\frac{4}{\Delta^2}\,)\,)$$

   8.5 Compute the acceleration, $\ddot{\mathbf{u}}_{T+\Delta}$, given by

$$\ddot{\mathbf{u}}_{T+\Delta} = (\,\mathbf{u}_{T+\Delta} - \hat{\mathbf{u}}_{T+\Delta}\,) * (\,\frac{4}{\Delta^2}\,)$$

   8.6 Compute the corrected velocity, $\dot{\mathbf{u}}_{T+\Delta}$, given by

$$\dot{\mathbf{u}}_{T+\Delta} = \hat{\dot{\mathbf{u}}}_{T+\Delta} + (\,(\,\mathbf{u}_{T+\Delta} - \hat{\mathbf{u}}_{T+\Delta}\,) * (\,\frac{2}{\Delta}\,)\,)$$

   8.7 Output the displacement and velocity at time $T+\Delta$ ( $\mathbf{u}_{T+\Delta}$ and $\dot{\mathbf{u}}_{T+\Delta}$ ).

   8.8 If the response at the end of next time step is desired, set $T = T + \Delta$ and go to step 8.1.

In the above steps, note that an object of a user defined type always precedes an over-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

loaded operator; for example, it is written $\dot{\mathbf{u}}_T * 3$ and not $3 * \dot{\mathbf{u}}_T$. The object that precedes an operator is the receiver of the message and in C++ overloaded operators can be defined only by classes and not for built in types.

*Reusability of Classes :*

The classes that must be defined in each step of the formulation are described below. The classes related to the finite element method are from or derived from the classes used in OPFI. Using inheritance mechanisms, the coding for the derived classes is just limited to patch the differences. The new classes that need to be defined for the dynamic analysis are *vector* and *matrix*; these classes must capture the mathematical concept of vector and matrix so that they become reusable classes for other applications such as different step-by-step solution methods that may be programmed with minimal effort in the future.

Class for Step 1: The input data for the extended program is made up of the data for OPFI and some additional data for dynamic analysis. Thus the *dynamic-input* class can be derived from OPFI's *input* class where the extension type of inheritance can be used to accommodate the data for initial conditions, time dependent load, time step, and number of time steps.

Classes for steps 2 and 2.1: The class for the object $\mathbf{k}_e$ is OPFI's *estif* (element stiffness). The base class for the object **K** is OPFI's *gstif* (global stiffness). The new derived class for **K**, call it *d-gstif*, must support three new methods: matrices addition { $\mathbf{K} + \mathbf{M}$ }, matrix-vector multiplication { $\mathbf{K} * \mathbf{u}$ }, and matrix-scalar multiplication { $\mathbf{M} * ( \frac{4}{\Delta^2} )$ }. A good strategy is to inherit these matrix properties from the generic *matrix* (described below) class using multiple inheritance so that the *gstif* from OPFI does not have to be altered and the derived *d-gstif* can be uniformly treated as *matrix* in other parts of the program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Classes for steps 3 and 3.1: The class for object $\mathbf{m}_e$, *emass* (element mass), can be derived from *Gpoint* (Gauss points), *Bmat* (B matrix), *Dmat* (D matrix) and *Shape* (Shape functions) classes that were used to derive the *estif* class in OPFI. The class for the object **M**, *gmass* (global mass), should be derived from the *matrix* class and support matrix addition { **K** + **M** }, matrix-constant multiplication { **M** * 6 }, matrix-vector multiplication { **M** * **u** }, factorization and solution. The derived properties of *gmass* include an operation for assembling the $\mathbf{m}_e$ using an algorithm identical to the direct stiffness method.

Classes for steps 4 through 8: The objects in steps 4 through 8 are instances of classes already described. The object $\tilde{\mathbf{K}}$ is a *d-gstif* and $\mathbf{u}$ , $\dot{\mathbf{u}}$ , $\ddot{\mathbf{u}}$ , and **F** with subscripts are all *vectors*. A new output class to collect all the responses at each time step for post processing by another module may be defined. However, if only the displacement is output as it is computed, the *output* class from OPFI can be used.

Class *vector*: The *vector* class for this program can be defined as a vector whose elements are real numbers. Three binary operations must be supported: the * operation where the argument is a scalar, the + operation where the argument is a vector and the − operation where the argument is a vector. Objects which are instances of *vector* may become arguments to other operations where the definitions of the operations are the implementation details of the message receiver. The access methods for elements in a *vector* must be efficient and versatile; for example, in many cases only few elements in a *vector* are non zero and efficient methods that return only these values and locations are needed in order to implement efficient operations that take these *vectors* as arguments. If the reusability of the *vector* class in the future is desired, the full mathematical concept of vector used in engineering must be implemented which includes methods for: 1. vector product yielding a matrix and 2. scalar product of two vectors yielding a scalar.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Also, a generic vector (a vector where the type of its elements can be user defined) is useful if vectors of varying types are needed.

Class *matrix*: The *matrix* class is needed as the base class to treat the objects **K**, **M** and $\tilde{\mathbf{K}}$ uniformly as *matrix* in the specification of the algorithm for Newmark's method. Thus, matrices addition, matrices subtraction, matrix-constant multiplication, matrix-vector multiplication, matrix factorization and solve operations must be supported. The derived classes *d-gstif* and *gmass* define the implementation details of the data structures; this is important because the matrices concerned are sparse, and mechanisms for storing only the non-zero elements are implemented at the derived class level. At the *matrix* level, the objects may be conveniently viewed as a square matrix that fully encapsulates the mathematical concept of a matrix. In C++, this can be implemented by declaring the full set of methods at the *matrix* level and declare those that depend on the implementation details of the data structures as *virtual*. The definitions of the *virtual* methods can be coded later at the derived class level.

Based on an object oriented finite element program that was developed as a teaching tool using the C++ language, this chapter presented a general guideline for object oriented development of engineering software. The guideline is based on levels of abstraction, which exist in most engineering problems. A substantial portion of the code produced following the guideline seems to be reusable in compiled form.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 5

# AN OBJECT ORIENTED DATA MODEL FOR ENGINEERING DATABASES

## 5.1. Introduction

This chapter reviews database concepts and proposes an object oriented data model for engineering databases.

The central database for integrated structural engineering systems is most likely an object oriented database [Powell 88a]. Research continues on a number of object oriented data models that share several features (see references [Fenves 89b; Kim 89; King 86; Lecluse 88; Lyngbaek 86]); many of these models are tied to the object oriented database that is built to support a specific application (e.g., integrated office systems, support for an object oriented programming environment, CAD, expert systems) but a general model or one suitable for engineering applications has not emerged yet.

## 5.2. Review of Database and Data Models

Computer science writers do not agree on how the term *database* should be written; some use it as one word as in this dissertation, some hyphenate the term (*data-base*), and others divide it into two words (*data base*). However, database systems have become an established field in the study of computer science with basic concepts. These concepts are reviewed in Section 5.2.1. Data models are treated separately in Section 5.2.2; traditional data models - the network, hierarchical, and relational models - are presented in Section 5.2.2.1 and some data model issues for engineering data are discussed in Section 5.2.2.2. Section 5.2.3 describes object oriented databases.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 5.2.1. Basic Database Concepts and Terminology

A *database* is a collection of stored *operational data* used by the application system of some *enterprise*. *Enterprise* is simply any reasonably large-scale commercial, scientific, technical or other operation. Any enterprise must maintain a lot of data about its operation and this is its *operational data*. Input and output data, which are transient, by themselves are not operational data [Date 83].

In general, operational data include some basic entities and relationships between these entities. It is important to note that relationships themselves can be entities and different ways of treating them produce different data models. The operational data stored in an integrated database provide the enterprise with *centralized* control of its operational data. The following list identifies the advantages that accrue from centralized control of the data:

1. The amount of redundancy in the stored data can be reduced.
2. Problems of inconsistency in the stored data can be avoided. If conflicting requirements exist, unbiased balance can be imposed.
3. The stored data can be shared among many users.
4. Standards can be enforced. This can simplify maintenance and data exchange with external enterprises.
5. Security restrictions can be selectively applied.
6. Data integrity can be maintained; i.e., the data values stored in the database must satisfy certain types of consistency constraints. In engineering, these constraints are generally physical; for example, data that represent volume, area or length cannot be negative.
7. Data independence can be provided; i.e., the organization and access strategy of data are not dependent on application modules. If a change in the organization of data or a better access strategy is implemented in the database, the application

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

modules are not affected.

Each user has some workspace and a language at his disposal. The language usually is a high level language such as FORTRAN, C or C++. The users' language includes a *data manipulation language* (DML), which is that subset of the language for transferring data between the data model and the workspace. A *data model* defines a set of data structures and operations which can be used for storage and manipulation of *data objects* in the database.

The DML usually includes syntax for the following:

1. The retrieval of information stored in the database.
2. The insertion of new information into the database.
3. The deletion of information from the database.

There are basically two types of DML: *procedural*, which requires a user to specify what data are needed and how to get them, and *nonprocedural*, which requires a user to specify what data are needed, without specifying how to get them.

A *query* is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called the *query language*. Although technically incorrect, it is common practice to use the terms *query language* and *data manipulation language* synonymously. In some older texts, *data sublanguage* (DSL) is used instead of a data manipulation language. Some query languages are more than a data manipulation language. These include part of the language that is appropriately called a *data definition language* (DDL). A *data definition language* is a special language where a *database schema* is specified by a set of definitions.

A *database schema* (called *database scheme* in some texts such as reference [Korth 86]) is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the overall design of the database. The collection of information stored in the database at a particular instant of time is called an *instance* of the database. The notion of *type definition* in programming languages corresponds to the concept of *database schema.* The notion of *value of a variable of a given type* corresponds to the concept of *database instance.*

There are several schemas in the database and based on the abstraction provided, they are divided into the following levels:

1. Physical Level: Describes *how* the data are actually stored in the database. This level need not concern application programmers.
2. Conceptual Level: Describes *what* data are stored in the database. This level is used by the *database administrator* (DBA), the one who decides what information is to be kept in the database.
3. View Level: This is the top level of abstraction where most of the unnecessary details are hidden. This level is used by application programmers.

An *index* is a special kind of stored file in the database where each entry consists of two values: a data value for some field of the indexed file and a pointer to a record of that file that contains the corresponding value. Index is primarily to speed up retrieval of data items in the database. Since it is not efficient to index all fields, the database administrator has to select which fields to index based on expected retrieval requests from users.

Applications usually deal with models that are not mapped directly to the data model but that are abstractions of the data model. Generally, these models are called *abstraction models* or *views.*

The *database management system* (DBMS) is the software which handles *all* access to the database. Any access request from a user is intercepted by the DBMS, which inspects the data model and then performs the necessary operations on the physical database. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DBMS is also responsible for applying the authorization checks and validation procedures.

In a simple case, each user's workspace acts as the receiving or transmitting area for all data transferred between the user and the database. In a more intricate case, such as in an integrated structural engineering system, the flow of data among program modules and the database has to be controlled. In civil engineering systems, four main approaches have been used:

1. Central Database Approach: The database is central to all applications and access is controlled by the DBMS. Application modules interact with the DBMS and interaction among the modules is through the database. This approach was used in NICE (Network of Interactive Computational Elements) [Felippa 81].

2. Coupled Database Approach: Program modules interact with a local database which is linked to the central database. Individual application modules can view the local database as if it is the central database and thus the database management system must provide location *transparency* to users; i.e., to a user, the location(local or central) of data is hidden. Coupled database is similar to distributed database systems [Stonebraker 83].

3. Superexecutive Approach: An executive program controls the flow of data between the application modules and the database. The executive program can assume some of the functions traditionally assigned to the DBMS. This approach was used in ICES (Integrated Civil Engineering System) [Roos 67].

4. Blackboard Approach : The program modules communicate data among themselves through the *blackboard* and also through the database. There is no direct link between the blackboard and the database [Fenves(S) 88].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 5.2.2. Data Models

A *data model* is a collection of conceptual tools for describing data, data relationships, data semantics, data organization, and consistency constraints. Associated with a data model is DDL (Data Definition Language), which the DBA (Database Administrator) uses for defining database schema and DML (Data Manipulation Language) which users utilize to access the database.

Data models can be grouped into *object-based logical models* and *record-based logical models*. Object-based logical models provide flexible structuring capabilities and allow one to specify data constraints explicitly. Record-based logical models require specification of both the overall logical structure of the database and a higher-level description of the implementation; however, record-based logical models do not provide facilities for specifying data constraints explicitly [Korth 86].

The network, hierarchical, and relational data models are record-based logical models that have been widely used. The relational model is *the* model of the 1980s. The network and hierarchical models were popular in the 1960s and 1970s.

#### 5.2.2.1. Network, Hierarchical, and Relational Models

*Network Model*

Data in the *network model* are represented by collections of *records* and *links*. A *record* is a collection of *fields*, each of which contains only one data value. A *link* represents a relationship between two records. In the network model, the links between records can form an arbitrary network or graph. This provides flexibility as a many-to-many (generally, there are one-to-one, one-to-many and many-to-many types of relationships) type of relationships can be modeled directly. However, the resulting network tends to be cumbersome

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and for this reason it is not widely used now. In addition, the associated DML(Data Manipulation Language) is procedural and the user has to know the details of the network.

In the late 1960s, several commercial database systems based on the network model emerged. These systems were studied extensively by the Database Task Group (DBTG) within the CODASYL (Conference on Data System Languages) group that earlier set the standard for COBOL. This study has resulted in the first database standard specification, called the CODASYL DBTG 1971 report [CODASYL 1971].

*Hierarchical Model*

Data in the *hierarchical model* are represented by collections of records and links, similar to that used in the network model. The hierarchical model differs from the network model in that records are organized as collections of trees. The associated DML is procedural and a retrieval has to follow the rigid hierarchy where each record can have one parent and a number of children records. Some structural engineering data are naturally hierarchical [Nicklin 87].

The most influential database system based on the hierarchical model is the Information Management System (IMS) developed in the late 1960s by IBM and Rockwell International for the Apollo moon landing program [IBM 78]. The hierarchical model was widely used until efficient implementations of relational databases became available in the early 1980s.

Network and Hierarchical data models do not provide independence between the data model and how the model is physically implemented. In the network model, operations are constrained to follow strict paths defined by the links and generally the links in the data model correspond to physical links implemented in the database. In the hierarchical model, the relationship between records must be an ordered tree, i.e., hierarchical. This is advantageous if the natural relationships among the records is hierarchical but when they

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are not, the concocted hierarchical model of data produces redundancy and hence a possibility of inconsistency.

*Relational Model*

The *relational model* represents the database as a collection of tables [Codd 70]. Each table is assigned a unique name. A record, or row, in a table represents a relationship among a set of values. Given a table, each field, or column, has a unique name within the table and data type (integer, real, characters, etc.). Although tables are an intuitive notion, there is a direct correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.

Tables in the relational model must have the following properties:

1. No two records in a table are identical. Each table has a *primary key* which is made up of one or more fields in a record and the value of the primary is used to distinguish different rows.
2. The ordering of records is arbitrary.
3. The ordering of fields is arbitrary. This is because a field is referred to by field name and not by relative position.
4. Every field within a relation is an atomic (nondecomposable) data item.

The DML for the relational model can be procedural or nonprocedural. The procedural language is based on relational algebra which includes five fundamental operations: select, project, Cartesian product, union, and set-difference. All of these operations produce a new relation as their result so operations can generally be applied to the result of an operation. The relational algebra is a procedural language because the user has to provide a sequence of operations that generates the answer to a query. The nonprocedural language is based on relational calculus. The relational calculus can be thought of as a notation for expressing the definition of a relation which is to be derived from the data model. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

theoretical foundation based on relational algebra and calculus on which the relational model is based is a major strength of the relational model.

The relational algebra and relational calculus are concise and formal languages that are inappropriate for casual users of a relational database. Therefore, commercial relational database system products require a more user-friendly language: Quel and SQL (Structure Query Language) are two widely used query languages available today. The first two relational database systems are System R, which was completed in 1979 by the IBM San Jose Research Laboratory [IBM 1982], and Ingres, an experimental relational database developed at the University of California at Berkeley which led to a commercial product with the same name [Stonebraker 1986]. Quel was introduced as the query language for the Ingres database system and SQL was introduced as the query language for System R. In 1986, SQL was adopted by the American National Standards Institute (ANSI) as the relational database query language.

The relational data model includes two integrity rules:

1. Entity Integrity: No field in a record that is part of the primary key can be null.
2. Referential Integrity: A record referred by another record must exist.

The purpose of the integrity rules are obvious; if entity integrity is not enforced, a record may not have a value for the primary key and thus will not be uniquely identifiable. If referential integrity is not enforced, a record may refer to another record that does not exist in the database. In commercial relational databases, the entity integrity is enforced but the referential integrity is generaly not enforced for efficiency reasons [Stonebraker 89].

When the relationship among data are complex, database design (or schema design) requires transformation of relations to what are known as *Normal Forms*. There are several normal forms and each requires satisfaction of certain constraints on the fields. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

involves analyzing functional dependence of data entities based on intended semantics of the data. Normal forms is an extensive subject and details may be found in reference [Date 83].

**5.2.2.2. Data Model Issues for Engineering Data**

General engineering data involve relationships that are complex and data entities that require elaborate data structures. Elaborate data structures as entries in the tables of the relational data model can dramatically hinder the performance of responses to the data manipulation language which usually operates on collections of tables. In addition, when relationships among data items are complex, the relational database schema have to rely on transformation of tables into normal forms; this process is error prone especially for complicated relations [Chen 78]. For these reasons, the existing relational model does not appear to be well suited for general engineering data.

To deal with complex data, a generation of ideas came with data models that are now grouped as *post-relational models* which include the functional data model, the semantic data model and, most recently, the object oriented data model. These are generally object-based logical record models that attempt to add *semantics* to the data model. One of the first one was the *entity-relationship model* [Chen 76] that is now widely used in database design.

The literature is full of ideas and models ; e.g., classes [Hammer 81], roles [Bachman 77], molecular objects [Batory 85], Is-a hierarchy [Smith 77], Part-of hierarchy [Katz 86], synonyms [Lohman 83], unique identifiers [Codd 79], extended relational model such as POSTGRES [Rowe 87], etc. However, most of these models are good for certain type of problems and somewhat deficient as a general data model. A comparison of some engineering data models is given in [Hardwick 87].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In general, an engineering data model must have the following basic features:

1. Capability to model complex relationships among data entities;
2. Capability to model general semantics in the data entities:
3. A set of constraints in the data model for integrity (e.g., referential integrity and entity integrity) of the database;
4. Some mechanisms to modify and extend the underlying database schema.

Consider the modeling of a slab in a building. A slab is physically connected to beams and girders. There may be many slab suppliers and different contractors may work on different parts of the building. In addition to the apparent physical properties of the slab, the model of a slab must include the relationships it may have with different types of data that represent suppliers, contractors, beams and girders. The set of relationships the model of slab must capture is necessarily complex. One way of reducing the complexities in the relationships among data and the representations of data entities is to impose constraints in the data model and have semantics in the data entities. In business applications, the database sheme rarely changes; only the volume of data fluctuates. In engineering, the initial user requirements of the engineering database is not likely to be complete or sufficient for the entire life cycle of the database. Thus the original database scheme may later become inappropriate and mechanisms to modify and extend the database scheme during the life cycle of the engineering database are essential.

### 5.2.3. Object Oriented Databases

Over the last few years, researchers have developed object oriented database systems to meet the needs of complex database applications [Kim 89]. Some of these systems are now appearing in the commercial market place.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

First of all, object oriented databases are *databases* for *objects*. As such, they provide features common in modern database systems. These features include the following:

1. Persistence: An object should be able to outlive the process that created it.
2. Concurrency: Many concurrent processes should be able to share an object that is persistent.
3. Resiliency: A database should be fault-tolerant in cases where the system fails.
4. Consistency: A database should contain consistent data.
5. Queries: Efficient access method should support the data model of the database.

The features of object oriented databases that distinguish them from others are due, in part, to their resemblance to object oriented programming languages. These features include the following:

1. Encapsulation (See also Chapter 2, Section 2.2.2): Method, which is a type of code that has access to the instance variables of the object, is part of data. A data item in the object oriented database is an object which includes a set of methods to encapsulate the object's state and behavior. This enables object oriented databases to have application semantics embedded in each object.
2. Extensibility: Object oriented databases provide tools for building extensions and one of the major technique for extension is inheritance. Various inheritance mechanisms (See Chapter 2, Section 2.3.1) exist but many cannot be implemented efficiently and others need to be restricted so that the objects' semantics as they are extended remain consistent.

What exactly is an object oriented data model cannot be defined yet. However, there are some common features among the data models that are reported as object oriented and they include the following:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. The data model has some *encapsulation* and *extensibility* features.

2. The data model is *identity-based.*

Identity-based means that reference to an object in the database is made through that object's *unique* identity, the *object identifier*. This implies that an object's identity remains invariant across all possible modifications of its state and enforcement of entity and referential integrity is implicit in the model.

Active research continues in object oriented database systems (e.g., ORION [Kim 89]; Gemstone [Maier 86]; EXODUS [Carey 88]; Iris [Fishman 89]) and how these systems evolve remains to be seen. The data models for these object oriented databases vary but most of the features in these models by computer scientists are to study the feasible implementations of object oriented concepts applicable to the data models. The semantic richness to represent engineering information is generally not emphasized in the models.

On the other hand, the so called data models presented by the engineering community are either too vaguely described to be called a data model or too specifically developed for an application and thus by definition cannot be a good data model. In addition, these data models generally lack some mechanism to organize the numerous objects in the database for views and implementation independent schemes for access. An object oriented data model presented in the following section suggests some solutions.

### 5.3. An Object Oriented Data Model

The object oriented data model described in this section is based on object oriented concepts presented in Chapter 2. A number of the model's features are as follows:

1. Encapsulation: Only the object's methods have access to its instance variables.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Extensibility: Inheritance used to define new classes and schema evolution for existing classes where class variables, instance variables and methods can be added or deleted are supported by the data model.
3. Uniformity: A class is treated as a special type of object which results in uniformity of the model where all activities of the database are via message passing schemes.
4. Organization and Access: Sets of objects and Aggregation objects [Smith 77] supported by the model can be used for views and organization of objects. Associative access limited to indexable methods (defined in Section 5.3.1.) is also part of the data model.

The data model presented here is intended to be the underlying data model for an object oriented database for engineering applications. The current state of development of object oriented databases allows reasonable support for most features presented, although some may require more efficient implementation techniques than that exist today. The rest of this section presents the specifics of the data model. Features considered to be an implementation detail are outside the scope of this dissertation and are not described.

The presentation is organized as follows:

1. Behavior of Classes and Objects : Describes Class-Creator (a unique and special object that creates Class-Objects) and Class-Object (one for each class).
2. Initial Definition of a Class : Specifies the requirements for and restrictions on the initial definition of each class.
3. Defining Classes: Describes how a Class-Object is instantiated by the Class-Creator to define a new class. Indexable method is also defined.
4. Behavior of a Class-Object : Describes the functions of Class-Objects in this data model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Aggregation, Sets, and Associative Access : Describes how the objects in the database can be conceptually organized and how index is used for accessing objects.
6. An Example : Clarifies the terms used in this presentation.

*1. Behavior of Classes and Objects*

The proposed database has a special object called the Class-Creator which accepts a message that contains the initial definition of a class. When this message is received, Class-Creator invokes one of its methods and instantiates an object which has the definition of the class. This object is called a Class-Object in the subsequent descriptions. A unique Class-Object exists for each class defined in the database.

A Class-Object is a special type of object that has the class's initial definition. The state of a Class-Object can change to represent evolution of the class definition. Objects are instantiated by sending messages to the Class-Object.

*2. Initial Definition of a Class*

Initial definition of a class must include the following:

1. Class Variables : For each class variable, specify the type and value. The type may be an intrinsic type defined for the database such as integer, real, characters or a class defined in the database. A class is defined if its Class-Object is in the database. The value of a class type is an object identifier.
2. Instance Variables : For each instance variable, specify the type. The type may be an intrinsic type defined for the database such as integer, real, characters or a class defined in the database.
3. Base Classes : These classes must be defined in the database. For each base class, specify whether the class (the class being defined) is to be treated as a member of the base class (this is for indexing purposes).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. A set of method definitions : The set must include one or more *creator* method. A creator method is invoked to instantiate an object in this class.

*3. Defining Classes*

When the initial definition of a class is sent to the Class-Creator object, the definition can be either rejected or accepted.

If the initial definition of a class is rejected, then one of the following reasons is returned to explain the rejection:

1. The classes used as type specification or as base classes are not defined in the database.
2. The object identifier used as a value is not in the database.
3. Method definition invokes a method that is not in the scope, where the scope includes methods defined for the class and its base classes.

If the initial definition of a class is accepted, the following events ensue:

1. A unique Class-Object for the class is added to the database.
2. For each method, indexability is determined.

A method is *indexable* if

1. The return type of the method is atomic and the type of that value is indexable, i.e., integer, real, characters, or other types where an orderly sequence can be defined; and
2. The execution of the method does not change the state of this or any other object. If the code of the method does not contain any class or instance variables as left hand value, then the state of the object is not changed by the method. This have to be checked for other methods that this method can invoke.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Objects in a class can only be indexed according to the value returned by an indexable method.

Two types of index can be formed:

1. Index for all objects that are instances of the class.
2. Index for all objects that are members of the class. Members include instances of the class and instances of the derived classes where it was declared that the objects of the derived class are members of the base class.

*4. Behavior of a* Class-Object

A Class-Object (at least one creator method must be included in the initial definition of the class) invokes a creator method to instantiate objects in its class. There can be many different creator methods for a Class-Object.

A Class-Object can receive a message that requests an addition or deletion of a class variable, an instance variable, or a method. This is the mechanism for database schema evolution supported by this data model. To ensure consistency of existing objects in the database the following conditions in each case must be satisfied :

1. Add a class variable: The variable name must be different from the instance and other class variable names. The type and value of the variable must also be provided.
2. Delete a class variable: There are no methods in the class whose code refers to the variable that is being deleted.
3. Add an instance variable: The variable name must be different from the class variable and other instance variable names. The type and default value of the variable must also be provided. The default value is this instance variable's value for all the objects in this class and its derived classes that existed in the database.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Delete an instance variable: There are no methods that refer to the variable.

5. Add a method: There are no methods in the class that has the same name. If the class has a base class, then the method name can not conflict with a method name in the base class; this is to preserve the behavior of existing objects.

6. Delete a method: Other methods in the class do not invoke the method that is to be deleted. If there are derived classes, the methods in the derived classes do not invoke the method that is to be deleted.

An object's class may evolve but throughout an object's entire life cycle, its Class-Object stays unique.

*5. Aggregation, Sets and Associative Access*

Aggregation is a powerful abstraction for database design [Smith 77]. It can also be characterized as a type of inheritance in an object oriented paradigm (See Chapter 2, Section 2.3.1). Aggregation type of objects needs database support to efficiently access its group of component objects. Clustering (physically placing component objects close together) can improve the accessing performance.

The main organizational tool for the data model is sets. When Class-Object is instantiated, a set is also formed. This set, called a Class-Set, holds all the objects in the database that are instances of the class. Since each object has a unique object identifier, the objects naturally form a set.

The set operations supported are union, intersection and subset. New sets that contain objects from different classes can be formed. An individual object may be treated equivalent to a set that contains that single object.

A subset of objects can be formed by identifying the objects by object identifiers. Another

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

method of forming a subset is with associative access.

An associative access contains the following items:

1. Class-Object
2. An indexable method (which returns *x*)
3. An operator (*op*) and
4. A constant (*c*)

The semantics of the above associative access is the following : for every object in the Class-Object, if *x op c* is true put that object in the set that is to be returned. Note that *x op c* must return *true* or *false* value.

*6. An Example*

Consider an object oriented database that will contain two classes: BEAM and COLUMN. Then two COLUMN objects, c1 and c1, and one BEAM objects b1 are created in the database. Using the names introduced in this section the following events will take place.

1 Initially, the Class-Creator is the only object in the database.

2 A user sends a message to Class-Creator that contains the definition of the BEAM class; Class-Creator invokes one of its methods and instantiates BEAM-Object which is a Class-Object. Similar events take place for COLUMN class.

3 There are three objects now in the data base; Class-Creator that can instantiate Class-Objects, BEAM-Object that can instantiate BEAM objects, and COLUMN-Object that can instantiate COLUMN objects.

4 A message is sent (the sender can be and object, user, or the database administrator) to BEAM-Object to create one BEAM object: b1.

5 Two messages are sent to COLUMN-Object to create two COLUMN objects, c1 and c2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The proposed model has many desirable features: everything is an object and message passing scheme is the only mechanism necessary for communication among objects, users and the database administrator. The resulting model is simple, consistent and object oriented. The power of the model, however, relies on the design of classes and this is dependent on applications. The model remains a theoretical model in that it is not implemented in an object oriented database management system. Although the model looks promising, only efficient implementation will make it practical.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# CHAPTER 6
# SUMMARY AND CONCLUSIONS

## 6.1. Summary

This dissertation has investigated programming languages for engineering, an object oriented finite element program, an object oriented development of engineering software, and an object oriented data model for engineering databases. The approach adopted for this study has generally emphasized the practical aspects (such as reusing available resources, availability of systems in various machines, current programming and engineering practices) rather than theoretical concepts. Overall, an object oriented paradigm is the unifying theme.

The C++ language is advocated in Chapter 3 as an appropriate language to write the next generation of engineering programs including the integrated structural engineering systems. The reusability of existing codes in Fortran and C and the uniform support for procedural and object oriented paradigms are considered as major merits of C++ as a programming language for engineering software.

An object oriented finite element program is developed using the C++ language. Based on the finite element program, a general guide for object oriented design and development of engineering software using C++ is outlined in Chapter 4. It is based on levels of abstraction and reusability of classes where the objective is to produce clear and modular code that can be maintained. The design of classes based on levels of abstraction produces reusable classes that can become base classes for other classes that may be needed in the maintenance stage of the software; this is an efficient reusability mechanism because a substantial portion of the existing code can be used in compiled form. Examples of object oriented techniques such as inheritance, polymorphism, overloading and late binding are



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

shown in the context of an object oriented finite element program.

An object oriented data model for engineering databases is presented in Chapter 5. The model treats classes as objects which make the model simple and uniform; message sending is the only mechanism necessary to create objects, delete objects, and change the class definitions. The scheme for organizing the objects in the database are supported by sets and aggregation objects. For practical implementation considerations, restrictions on indexing and how class definitions can change are included in the model.

## 6.2. Recommendations for Further Research

Recently, both in academia and industry, there have been many theoretical investigations and developments of models for processes and tasks that are targets for an integrated structural engineering system [Abdalla 89; An-Nashif 89; Sauce 89]. This study has focused on fundamental tools for developing integrated structural engineering systems and other generally large engineering application programs. The object oriented software design method and the object oriented data model presented in this dissertation are general and fundamental tools for developing an integrated structural engineering system.

The recommendations for further research listed below are concerned with developing a prototype of an integrated structural engineering system. The prototype should help to understand whether these systems will be just a reliable organizational tool which makes current structural engineering practices more efficient or whether these systems can change the inherent inefficiencies in the organization of current practices and provide a better structure where various tasks and processes can interact in an integrated environment.

Some of the topics recommended for further research are as follows:

1. Implement the object oriented data model presented in Chapter 5 on an object

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

oriented database that provides the conventional database supports for objects such as persistence, security, crash recovery and concurrency. First, the syntax and grammar for data manipulation and data definition languages for the model must be designed; since the data model is defined, the emphasis here should be in choosing clear, meaningful, and coherent words and symbols for the languages. The queries take forms that are similar to the ones in the relational data model and thus many query optimizing techniques well established for the relational databases should be applicable.

2. The object oriented data model presented in Chapter 5 has features that are designed to support the semantic richness of data required by engineering applications. How to design the database schema with the model for these applications needs to be investigated. Designing the object classes and organizing the objects for different applications are some of the topics to be addressed.

3. Select and develop in detail a model for processes or tasks in structural engineering that can be integrated (e.g., Component-Connection Model for buildings [Powell 88a; Powell 88b], Multilevel-Selection-Development model for structural design [Sauce 89], etc.). Clearly identify the kinds of activity the model can support.

4. User interface is important for any software system. How to present the diverse elements in an integrated environment to a user is a difficult problem that can be a critical factor in determining the success or failure of an integrated system. Fundamental research in this area for engineering application programs are necessary.

5. Integration of object oriented programming languages and object oriented databases is a research area in computer science. This dissertation has implicitly assumed that there is a barrier between the applications and the database; i.e., an object oriented database has been treated as a separate component that can be interfaced to the rest of the system. An alternative object management system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where objects from databases and programs are treated uniformly may be a better underlying system for certain types of integrated structural engineering systems.

### 6.3. Concluding Remarks

Programming languages, methods of program development and even data models have characteristics common with religion. Many reasons people have for using a particular system or method are not based on rational principles but based rather on a strong belief typically biased towards the first system they ever learned.

In structural engineering, there has been a desire for quite some time to move away from Fortran, a language still used to write most structural engineering application programs. As programs became larger to meet today's more complex needs of clients, interest in database systems has also been increasing. However, a new dominant language to replace Fortran does not yet exist and use of a database in engineering applications is still not common. Several engineering systems have used relational databases but they have only indicated the limitation of the relational data model for engineering applications.

This dissertation has advocated use of C++, a guideline for object oriented development of engineering software, and an object oriented data model to describe engineering data in object oriented databases. These are implementation tools that structural engineers may choose to implement their software after the analysis of user requirements or a model for an integrated structural engineering system has been completed. Theoretically, software systems can be implemented with any tool, but often the quality and functions of a system depend on particular tools used and thus these are critical choices. For those contemplating implementation of an integrated structural engineering system, the ultimate contribution of this study is to aid in making rational choices for their project, even when the particular model and style advocated in this dissertation are rejected.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**REFERENCES**


[Abdalla 89] Abdalla, G.A., "Object-Oriented Principles and Techniques for Computer Integrated Design," Ph.D. Dissertation, Department of Civil Engineering, University of California at Berkeley, CA, 1989.

[Ada 83] *Ada Programming Language*, USA Military Standard, ANSI/MIL-STD-1815A, February 1983.

[Ada Letters 90] *ACM Ada Letters*, Vol. X, No. 3, ACM Press, New York, NY, Winter 1990.

[Anderson 64] Anderson, C., *An Introduction to ALGOL 60*, Addison-Wesley, Reading, MA, 1964.

[An-Nashif 89] An-Nashif, H.N., "Automated Structural Analysis for Computer Integrated Design," Ph.D. Dissertation, Department of Civil Engineering, University of California at Berkeley, CA, 1989.

[ANSI 85] ANSI Technical Committee X3J11, *Preliminary Draft Proposed Standard - The C Language*, X3 Secretariat, 311 First Street NW, Suite 500, Washington DC 20001, 1985.

[Augensten 79] Augensten, M., *Data Structures and PL/I Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Bachman 77] Bachman, C., Daya, M., "The Role Concept in Database Models," *Proceedings, 1977 VLSI Conference*, Tokyo, Japan, October 1977.

[Backus 81] Backus, J., "The history of FORTRAN I, II and III" in *History of Programming Languages*, edited by Weselblat, R.L., Academic Press, New York, NY, 1981.

[Bathe 82] Bathe, K.J., *Finite Element Procedures in Engineering Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 499-506.

[Batory 85] Batory, D., Kim, W., "Modeling Concepts for VLSI CAD Objects," IACM-TODS, September 1985.

[Baugh 89] Baugh, J.W., *Computational Abstractions for Finite Element Programming*, Ph.D. Dissertation, Department of Civil Engineering, Carnegie-Mellon University, September 1989.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Becker 86] Becker, E.B., Carey, G.F., Oden, J.T., *Finite Elements*, Vol. 1-6, Prentice-Hall, Englewood Cliffs, NJ, 1981-1986.

[Bell 87] Bell, D., Morrey, Pugh, J., *Software Engineering, A Programming Approach*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Belytschko 83] Belytschko, T., Hughes, T.J.R., (Editors), *Computational Methods for Transient Analysis*, Vol. 1, North-Holland, New York, NY, 1983.

[Bergman 90] Bergman, N.J., "Three faces of Smalltalk," *Computer Languages*, Vol. 7, No. 4, April 1990, pp. 87-91.

[Berry 88] Berry, J., *C++ Programming*, The Waite Group, Inc., Howard W. Sams & Company, Indianapolis, IN, 1988.

[Birtwistle 71] Birtwistle, G., et al., *SIMULA BEGIN*, Studentlitteratur, Lund, Sweden, 1971.

[Bobrow 83] Bobrow, D., Stefik, M., *The LOOPS Manual*, Xerox PARC, Palo Alto, CA, 1983.

[Boeing 82] Boeing Computer Services Co., *RIM - Relational Information Management System (Version 5.0)*, Boeing Computer Services Co., Seattle, WA, 1982.

[Booch 86] Booch, G., "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211-221.

[Cannon 80] Cannon, H.I., *Flavors*, Technical Report, MIT Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1980.

[Carey 88] Carey, M.J., DeWitt, D.J., Vandenberg, S.L., "A Data Model and Query Language for EXODUS," *ACM SIGMOD International Conference on Management of Data*, Chicago, Il, June 1988, pp. 413-423.

[Cassel 83] Cassel, D., *The Structured Alternative: Program Design, Style, and Debugging*, Reston Publishing Company, Inc., Reston, VA, 1983.

[Chen 76] Chen, P.P.S., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transaction on Database Systems*, Vol. 1/No. 1, January 1976, pp. 9-36.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Chen 78] Chen, P.P.S., "The entity-relationship approach to logical database database design," Monogram 6, *QED Information Sciences*, Wellesley, MA, 1978.

[Clough 75] Clough, R.W., Penzien, J., *Dynamics of Structures*, McGraw-Hill, New York, NY, 1975.

[CODASYL 71] CODASYL Data Base Task Group, "CODASYL Data Base Task Group April 71 Report," ACM, New York, NY, 1971.

[Codd 70] Codd, E.F., "A Relational Model for Large Shared Databanks," *Communication of The ACM*, Vol. 13, No. 6, 1970, pp. 377-390.

[Codd 79] Codd, E.F., "Extending Database Relations to Capture More Meaning," *ACM-TODDS*, December 1979.

[Colmerauer 73] Colmerauer, A., et al., *Un Systeme de Communication Homme-machie en Francais*, Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, France, 1973.

[Courant 43] Courant, R., "Variational Methods for the Solution of Problems of Equilibrium and Vibration," *Bulletin of American Mathematics Society*, Vol. 49, 1943, pp. 1-43.

[Cox 86] Cox, B.J., *Object Oriented Programming, An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.

[Craine 81] Craine, L., et al., *PDA/PATRAN-G Users Guide*, PDA Engineering, Santa Ana, CA, 1981.

[Date 83] Date, C.J., *An Introduction to Database Systems*, Third Edition, Addison-Wesley, Reading, MA, 1983.

[Davis 78] Davis, G.B., Hoffman, T.R., *FORTRAN, a Structured, Disciplined Style*, McGraw-Hill, New York, NY, 1978.

[Dawson 89] Dawson, J., "A Family of Models", *Byte*, Vol. 14/No. 9, September 1989, pp. 277-286.

[Dijkstra 76] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Felippa 81] Felippa, C.A., "Architecture of a Distributed Analysis Network for Computational Mechanics," *Computers and Structures*, 13(1-3), 1981, pp. 405-413.

[Fenves 89a] Fenves, G.L., CE290C Course Notes, Department of Civil Engineering, University of California at Berkeley, CA, 1989.

[Fenves 89b] Fenves, G.L., "Object Oriented Models for Engineering Data," *Proceedings*, ASCE Sixth Conference on Computing in Civil Engineering, Atlanta, Georgia, September 1989.

[Fenves 90] Fenves, G.L., "Object Oriented Programming for Engineering Software Development," *Engineering with Computers*, Vol. 6, 1990, pp. 1-15.

[Fenves(S) 88] Fenves, S.J., Flemming, U., Hendrickson, C., Maher, M.L., Schmitt, G., "An Integrated Software Environment for Building Design and Construction," *Proceedings of the Fifth ASCE Conference on computing in Civil Engineering: Microcomputers to Supercomputers*, Alexandria, VA., March 1988, pp. 21-32.

[Fishman 89] Fishman, D.H., et al., "Overview of the Iris DBMS," *Object Oriented Concepts, Database, and Applications*, edited by Kim, W., Lochovsky, F.H., ACM Press, New York, NY, 1989.

[Garett 89] Garrett Jr., J.H., Basten, J., Breslin, J., Anderson, T., "An Object-Oriented Model for Building Design and Construction", *Proceedings*, Sessions Related to Computer Utilization at Structures Congress '89, San Francisco, CA, May 1989, pp. 332-341.

[Global 87] XX Draft Proposed American National Standard For Fortran, Global Engineering Documents Inc., Santa Ana, CA, October 1987.

[Goldberg 83] Goldberg, A., Robson, D., *Smalltalk-80: The Language and the Implementation*, Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984.

[Hammer 81] Hammer, M., McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM-TODS*, September 1981.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Hardwick 87] Hardwick, M., Spooner, D.L., "Comparison of Some Data Models of Engineering Objects," *IEEE Computer Graphics and Applications*, Vol. 7/No. 3, 1987, pp. 56-66.

[Hogg 85] Hogg, J., Nierstrsz, O.M., Tsichritzis, D.C., "Office Procedures," in *Office Automation: Concepts and Tools* (edited by Tsichritzis, D.C.), Springer-Verlag, Heidelberg, 1985, pp. 137-166.

[Holtz 88] Holtz, N.M., Rasdorf, W., "An Evaluation of Programming Languages and Language Features for Engineering software Development," *Engineering with Computers*, Vol. 3, 1988, pp. 183-199.

[Howard 89] Howard, H.C., Levitt, R.E., "Linking Design Data with Knowledge-Based Construction Systems", CIFE Flagship Project Proposal, Department of Civil Engineering, Stanford University, Palo Alto, CA, October 1989.

[Hughes 87] Hughes, T.J.R., *The Finite Element Method*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[IBM 78] IBM Corporation, *Information Management System/Virtual Storage General Information*, IBM Form No. GH20-1260, SH20-9025, SH20-9026, SH90-9027, 1978.

[IBM 82] IBM Corporation, *SQL/Data System Terminal Users Guide*, IBM Form No. SH24-5017-1, 1982.

[Ichbiah 79] Ichbiah, J.D., et al., *Rationale for the Design of the Ada Programming Languages*, SIGPLAN Notices, June 1979.

[Katz 85] Katz, R.H., *Information Management for Engineering Design*, Springer-Verlag, 1985.

[Katz 86] Katz, R.H., Chang, E., Bhateja, R., "Version Modeling Concepts for Computer-Aided Design Databases," Report No. UCB/CSD 86/270, EECS, University of California at Berkeley, CA, November 1986.

[Keene 89] Keene, S.E., *Object-Oriented Programming in Common LISP : a Programmer's Guide to CLOS*, Addison-Wesley, Reading, MA, 1989.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Keirouz 87] Keirouz, W.T., Rehak, D.R., Oppenheim, I.J., "Development of an Object-Oriented Domain Model for Constructed Facilities," Report No. EDRC-12-10-87, Carnegie Mellon University, Pittsburgh, PA, 1987.

[Kernighan 78] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kernighan 87] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Kim 1989] Kim, W., Lochovsky, F.H. (Editors), *Object -Oriented Concepts, Databases, and Applications*, ACM Press, New York, NY, 1989.

[King 86] King, R., "Database Management System Based on an Object-Oriented Model," *Expert Database Conference*, edited by Kerschberg, L., Benjamin/Cummings, Menlo Park, CA, 1986.

[Knuth 74] Knuth, D.E., "Structured Programming with Go To Statements," *Computing Surveys*, ACM, Vol. 6, No. 4, 1974, pp. 261-301.

[Kohnke 78] Kohnke, P.C., *ANSYS Theoretical Manual*, Swanson Analysis Systems, Inc., Houston, PA, May 1978.

[Korth 86] Korth, H.F., Silberschatz, A., *Database System Concepts*, McGraw-Hill, New York, NY, 1986.

[Kruse 84] Kruse, R, *Data Structures and Program Design*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[Ladd 89] Ladd, S.R., "OOPing in public", *Computer Languages*, Vol. 7, No. 2, February 1990, pp. 127-133.

[Lecluse 88] Lecluse, C., Richard, P., Velez, F., "O2, an Object-Oriented Data Model," *ACM Conference on the Management of Data*, 1988, pp. 424.

[Lippman 89] Lippman, S., *C++ Primer*, Addison-Wesley, Reading, MA, 1989.

[Liskov 77] Liskov, B., et al., "Abstraction Mechanisms in Clu," *CACM* Vol. 20, No. 8, August 1977, pp. 564-576.

[Logcher 71] Logcher, R.D., et al., *ICES STRUDL-II, Engineering User's Manual*, Department of Civil Engineering, MIT, Cambridge, MA, 1971.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Maier 87] Maier, D., Stein, J., Otis, A., Purdy, A., "Development of an Object-Oriented DBMS," *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1986.

[McLean 81] Mclean, D.M. (editor), *MSC/NASTRAN Programmer's Manual*, MSR-59, The MacNeal-Schwendler Corporation, Los Angeles, CA, June 1981.

[Milner 84] Milner, R., "A Proposal for Standard ML," *ACM Symposium on Lisp and Functional Programming*, August 1984, pp. 184-197.

[NASA 79] *The NASTRAN User's Manual (Level 17.5)*, NASA SP223 (3&4), COSMIC, 1979.

[Neeley 89] Neeley, D., "Integrated Solutions," *Cadence*, October 1989, pp. 109-110.

[Newmark 59] Newmark, N.M., "A Method of Computation for Structural Dynamics," *Journal of the Engineering Mechanics Division*, ASCE, 1959, pp. 67-94.

[Nicklin 87] Nicklin, P.J., Powell, G.H., Hollings, J.P., "Hierarchical Data Management for Structural Analysis," *Engineering with Computers*, Vol. 1, 1987, pp. 135-143.

[Nikhil 88] Nikhil, R.S., *ID Reference Manual*, Computation Structures Group Memo 284, MIT Laboratory for Computer Science, March 1988.

[Plauger 90] Plauger, P.J., "The Politics of Standards," *Computer Language*, Vol. 7, No. 2, 1990, pp. 17-22.

[Powell 88a] Powell, G.H., Bhateja, R., "Data Base Design for Computer-Integrated Structural Engineering," *Engineering with Computers*, Vol. 4, 1988, pp. 135-143.

[Powell 88b] Powell, G.H., Bhateja, R.,Abdalla, G., An-Nashif, H., Martini, K., Sauce, R., "A Database Concept for Computer Integrated Structural Engineering Design," *Proceedings*, ASCE Fifth Conference on Computing in Civil Engineering, Alexandria, Virginia, March 1988.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Rasdorf 85] Rasdorf, W.J., Salley, G.C., "Generative Engineering Databases - Towards Expert Systems," *Computers and Structures*, vol. 20, 1985, pp. 11-15.

[Rehak 89] Rehak, D.R., Baugh, J.W., "Implementation of a Finite Element Programming System - A Declarative Approach," *Computer Utilization in Structural Engineering*, ASCE Structures Congress '89, San Francisco, CA, May 1989, pp. 91-100.

[Roos 67] Roos, D., *ICES System Design*, Second Edition, Cambridge, MA., 1967.

[Rowe 86] Rowe, L., CS160 Course Notes, Department of Electrical Engineering and Computer Science, University of California at Berkeley, CA, 1986.

[Rowe 87] Rowe, L.A., Stonebraker, M.R., "The POSTGRES Data Model," *Proceedings, Thirteenth International Conference on Very Large Databases*, Brighton, U.K., 1987.

[Sauce 89] Sauce Jr., S., "A Model of the Design Process for Computer Integrated Structural Engineering," Ph.D. Dissertation, Department of Civil Engineering, University of California at Berkeley, CA, 1989.

[Schmucker 86] Schmucker, K., *Object-oriented Languages on the Macintosh*, Apple Press, 1986.

[Smith 77] Smith, J., Smith, D., "Database Abstractions: Aggregation and Generalization," *ACM-TODS*, July 1977.

[Smith 86] Smith, B., Wellington, J., *Initial Graphics Exchange Specification (IGES) Version 3.0*, National Bureau of Standards, NBSIR 86-3359, Gaithersburg, Maryland, April 1986.

[Smith 90] Smith, J.D., *Reusability & Software Construction: C & C++*, John Wiley & Sons, New York, NY, 1990.

[Sommerville 89] Sommerville, I., *Software Engineering*, 3 ed., Addison-Wesley, Reading, MA, 1989.

[Sterling 86] Sterling, L., Shapiro, E., *The Art of Proglog*, MIT Press, Cambridge, MA, 1986.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Stonebraker 83] Stonebraker, M., et.al., "Performance Analysis of Distributed Data Base Systems," *Proc. Third Symposium on Reliability in Distributed Software and Database Systems*, Clearwater, FL., 1983.

[Stonebraker 86] Stonebraker, M., (Editor), *The Ingres Papers*, Addison Wesley, Reading, MA, 1986.

[Stonebraker 90] Stonebraker, M., CS286 Course Notes, Department of Electrical Engineering and Computer Science, University of California at Berkeley, CA, 1990.

[Stroustrup 86] Stroustrup, B., *The C++ Programming language*, Addison-Wesley, Reading, MA, 1986.

[Stroustrup 87] Stroustrup, B., "Multiple Inheritance for C++," *Proceedings of the Spring '87 EUUG Conference*, Helsinki, May 1987.

[Stroustrup 88a] Stroustrup, B., "What is Object-Oriented Programming," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 10-20.

[Stroustrup 88b] Stroustrup, B., "Parameterized Types for C++," *C++ Conference Proceedings*, USENIX Association, Denver, CO, October 1988, pp. 1-18.

[Taylor 77] Taylor, R.L., "Computer Procedures for Finite Element Analysis," Chapter 24 in *The Finite Element Method* by O.C. Zienkiewicz, 3rd Edition, McGraw-Hill Book Co., London, 1977.

[Taylor 87] Taylor, R.L., CE222 Course Notes, Department of Civil Engineering, University of California at Berkeley, CA, Fall 1987.

[Turner 56] Turner, M.L., Clough, R.W., Martin, H.C., and Topp, L.J., "Stiffness and Deflection Analysis of Complex Structures," *Journal of Aeronautical Sciences*, Vol. 23, No. 9, 1956, pp. 805-823.

[USENIX 88] *C++ Conference Proceedings*, USENIX Association, Denver, CO, October 1988.

[Weiner 88] Weiner, R.S., Pinson, L.J., *An Introduction to Object Oriented Programming and C++*, Addison-Wesley, Reading, MA, 1988.

[Wilensky 84] Wilensky, R., *LISPcraft*, W.W.Norton & Company, 1984.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Wilson 70] Wilson, E.L., "SAP - A General Structural Analysis Program," Report No. UCB/SESM-70/20, University of California at Berkeley, CA, September 1970.

[Wilson 90] Wilson, E.L., Habibullah, A., "SAP90 - A Program for the Static and Dynamic Finite Element Analysis of Structures," Computer & Structures, Inc., 1918 University Avenue, Berkeley, CA 94704, 1990.

[Wirth 71] Wirth, N., "The Programming Language PASCAL," *Acta Infomatica* 1, 1971, pp. 35-63.

[Wirth 77] Wirth, N., "Modula: A language for multiprogramming," *Software-Practice and Experience* 7, 1977, pp. 3-35.

[Wirth 88] Wirth, N., *Programming in Modula-2*, 4th Ed., Springer-Verlag, Berlin, 1988.

[Yoon 89] Yoon, C., "OPFI - Object Oriented Program for Finite Element Instruction," Report No. UCB/SESM-89/02, University of California at Berkeley, CA, February 1989.

[Zienkiewicz 89] Zienkiewicz, O.C., Taylor, R.L., *The Finite Element Method*, 4th edition, McGraw-Hill Book Co., London, 1989.

[Zortech 89] *Zortech C++ Compiler V2.0: Compiler Reference*, Zortech Inc., Arlington, MA, 1989.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**APPENDIX : Declarations of Classes in OPFI**

1. input
2. output
3. gstif (global stiffness)
4. Shape (shape functions)
5. Stress (stress matrix)
6. Stif_mat (stiffness matrix)
7. Dmat (elasticity matrix)
8. Gpoint (Gauss weights and points)
9. Bmat (strain matrix)
10. estif (element stiffness)

```
/*
**   DECLARATIONS FOR CLASS input
*/
class input { double* coor     ;
              int*    nelem    ;
              int*    nboun    ;
              double* dload    ;
              char*   mate_name;
              double* dmate    ;
public: // PUBLIC FUNCTIONS FOR CLASS input
  input(int dim, int dof, int nen, int npt, int nel, int nbc,
        int nld, int nms, int nmt) ;
                          //    ARGUMENT          RETURNED VALUE
  double         x(int); //  node number          x coordinate
  double         y(int); //  node number          y coordinate
  double         z(int); //  node number          z coordinate
  double  xyz(int,int); //dim, node number     dim's coordinate
                          //  dim =1 for x, 2 for y, and 3 for z
  int*       elem(int); // element number element incidence[nen]
  int    boun_nod(int); //   bc number           node number
  int    boun_con(int); //   bc number       code: 0=fixed;
                          //                      n=nth dof fixed
  int    load_nod(int); //  load number          node number
  int    load_dof(int); //  load number dof of load: 1=x,2=y,3=z
  double load_val(int); //  load number          load value
  char*  mat_name(int); //property number     property value
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  double
   mate_val(char*,int); //    material      material property
};                      //   name,  set          value

/*
**   DECLARATIONS FOR CLASS output
*/
class output { input*  Ip;
               gstif*  Kp;
               double* aa;
               int     na;
public: // PUBLIC FUNCTIONS FOR CLASS output
  output(input& I, gstif& K);
  double* aux(int) ;     // return auxiliary data for element n
  void displ(int=0);     // print  nodal displacements
};

/*
**   DECLARATIONS FOR CLASS gstif
*/
class gstif {  int*     id ;
               int      neq;
               double*  b  ;
               double*  d  ;
               double*  u  ;
               int*     lm ;
               int*     jp ;
               input*   Ip ;
public: // PUBLIC FUNCTIONS FOR CLASS gstif
  gstif(input&)              ;
  int eq(int,int)            ; // return equation number starting
                               //   from 1, given (dof, coor)
  double* dl_vec()           ; // return load vector b
  void add(double,int,int)   ; // add(a,i,j) adds a to A(i,j)
  gstif& operator+(estif&)   ; // assemble estif to gstif, K= K+KE
  void load()                ; // form load vector in b
  void solve()               ; // solve Ax=f (x=b,f=b,A->d,u,jp)
};                             //    A is symmetric

/*
**   DECLARATIONS FOR CLASS Shape
*/
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
class Bmat;
class Shape{  char* s      ;
public: // PUBLIC FUNCTIONS FOR CLASS Shape
  Shape()                  ; //     ARGUMENT      RETURNED VALUE
  char* shp(int)           ; //  integer(1-nen)   shape function
};

/*
**   DECLARATIONS FOR CLASS Stress
*/
class Stress { double* s   ; // column major
public: // PUBLIC FUNCTIONS FOR CLASS Stress
  Stress()                 ;
  void store()             ; // **not implemented
  double* matrix()         ; // return s
  Stress& operator*(double); // * operator, ->  S = S*a
};                           //   Stress& = Stress*double

/*
**   DECLARATIONS FOR CLASS Stif_mat
*/
class Stif_mat { double* d ;
                 double* u ;
public: // PUBLIC FUNCTIONS FOR CLASS Stif_mat
  Stif_mat()               ;
  double* dstif()          ; // return d
  double* ustif()          ; // return u
  void    init()           ; // initialize d and u to zero
  Stif_mat&                  // + operator, ->  eK=eK+gK
       operator+(Stif_mat&); //  Stif_mat&=Stif_mat+Stif_mat&
};

/*
**   DECLARATIONS FOR CLASS Dmat
*/
class Dmat { double* d_t    ; // row major, i.e. transposed
             input*   Ip    ;
public: // PUBLIC FUNCTIONS FOR CLASS Dmat
  Dmat(input&)              ;
  double* matrix_t()        ; // return d_t, -row major
  Stress operator*(Bmat&)   ; // * operator, -> S = D*B
};                            //    Stress = Dmat*Bmat&
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*
**   DECLARATIONS FOR CLASS Gpoint
*/
class Gpoint { int      ng     ;
               double* g       ;
public: // PUBLIC FUNCTIONS FOR CLASS Gpoint
  Gpoint()                     ; //  ARGUMENT         RETURNED VALUE
  int  points()                ; //                     gauss points
  double wt(int)               ; // gauss point         gauss weight
  double xl(int)               ; // gauss point       local x coordinate
  double yl(int)               ; // gauss point       local y coordinate
  double zl(int)               ; // gauss point       local z coordinate
  double xyzl(int,int)         ; // dim,  point       local dim coordinate
};

/*
**   DECLARATIONS FOR CLASS Bmat
*/
class Bmat{ int*       bn      ;
            double*     b      ; // column major, trans()->row major
            double    jac      ;
            input*     Ip      ;
            double* coor       ; // 1-d, global coordinates of nodes
            char*     shp      ; // 1-d, each shape is  char[100]
            char*     drv      ; // 1-d, each derivative is char[300]
public: // PUBLIC FUNCTIONS FOR CLASS Bmat
  Bmat(input&)                 ; // allocate storage
                                 //  RETURNED VALUES
  double* matrix()             ; //          b
  double jacobian()            ; //         jac
  double x(int i)              ; // x coordinate of node i
  double y(int i)              ; // y coordinate of node i
  double z(int i)              ; // z coordinate of node i
  double xyz(int i,int j)      ; // dim i coordinate of node j
  char* shape(int)             ; // shape function for node int
  char* deriv(int i, int j)    ; // dim i derivative of shape j
  int ddim(int i,int j)        ; // derivative dimension of B(ij)
                                 //   EFFECTS
  void set_val(Shape&)         ; // set shp and drv values
  void set_xyz(double**)       ; // set xyz values
  void form(Gpoint&,int)       ; // form B(G,G.point_number) in b
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  void trans()                  ; // form B transposed, in row major
  Stif_mat operator*(Stress); // usage: gK = B*S
};

/*
**   DECLARATIONS FOR CLASS estif
*/
class estif { Stif_mat eK  ; // eK,G,D,Shp,B initialized first
              Gpoint    G  ;
              Dmat      D  ;
              Shape   Shp  ;
              Bmat      B  ;
              int      el  ;
              input*   Ip  ;
public: // PUBLIC FUNCTIONS FOR CLASS estif
  estif(input&)            ; //   RETURNED VALUES
  double* stif_d()         ; // diagonal element stiffness
  double* stif_u()         ; // off-diagonal element stiffness
  int     elem()           ; // element number
                             //   EFFECT
  void    form(int)        ; // form element stiffness(element no.)
};
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.