

Object Oriented Programming for Reinforced Concrete Design

by

Ajay B. Kulkarni

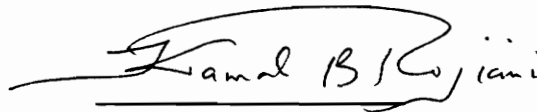
Thesis submitted to the Faculty of the
Virginia Polytechnic Institute & State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

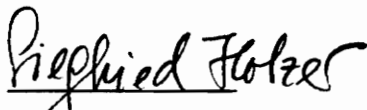

in

Civil Engineering

Approved:



K. B. Rojiani


S. M. Holzer
D. A. Garst

May, 1993

Blacksburg, Virginia

C.2

562
4725
1993
K855
C.2

/

Object Oriented Programming for Reinforced Concrete Design

by

Ajay B. Kulkarni

Committee Chairman: Prof. Kamal B. Rojiani

Charles E. Via, Jr. Department of Civil Engineering

Virginia Polytechnic Institute and State University

Abstract

The use of the object oriented programming approach in developing applications for the analysis and design of reinforced concrete structures is discussed. Two object oriented programming languages, Actor and Borland C++ for Windows were used to develop several applications. Actor is a pure object oriented programming language while C++ is a hybrid object oriented programming language. A simple program for computing the flexural capacity of reinforced concrete tee beams was developed in both languages. A second program for the analysis and design of reinforced concrete continuous beams was developed in Actor and C++. This application is representative of a practical structural engineering application and has both analysis and design components. The procedures and techniques used in the development of this application can easily be applied to the development of other structural engineering applications. A third program for the design of simply supported beams was also developed in Actor.

The advantages and disadvantages of object oriented programming for structural engineering application development were studied. It was found that object oriented programming has significant benefits. However, these benefits can only be utilized if careful thought is given during the program development stage. There is also some overhead associated with object oriented programming. A comparative study of the two programming languages: Actor and Borland C++ was also performed.

Acknowledgments

I wish to express my sincere gratitude to my advisor Dr. Kamal B. Rojiani. His patience, insight and experience in structural engineering and computing helped me surmount the uphill task of learning and implementing object oriented programming. I would like to thank Dr. S. M. Holzer for his enthusiasm and advice from time to time during the course of my graduate studies and to Prof. D. A. Garst for his willingness to review the thesis and to serve on the committee. I would also like to thank Dr. T. Kuppusamy for his advice and help.

Finally, I would like to express deep gratitude for my parents without their support this Master's study would not have been possible and I am deeply indebted to them. I would also like to express special thanks to my sister and brother for their constant encouragement and invaluable moral support.

Table of Contents

Acknowledgements.....	iii
List of Figures	vii
List of Tables.....	x
 Chapter 1	 1
Introduction.....	1
1.1 Introduction	1
1.2 Purpose and Scope.....	2
1.3 Organization.....	3
 Chapter 2	 4
An Overview of Object Oriented Programming	4
2.1 Introduction	4
2.2 Programming Methodologies.....	4
2.3 Object Oriented Programming	5
2.4 Elements of Object Oriented Programming	6
2.4.1 Classes and Inheritance	6
2.4.2 Object.....	7
2.4.3 Message Passing	8
2.5 Object Oriented Programming: Why is it needed ?	8
2.6 Object Oriented Programming Languages.....	11
2.7 Shortcomings of Object Oriented Programming	12
2.8 Literature Review.....	14
 Chapter 3	 18

Overview of Actor and C++.....	18
3.1 Introduction	18
3.2 Actor	18
3.2.1 Actor Programming	19
3.2.2 The Actor Environment	23
3.2.3 The WhiteWater Resource Toolkit.....	29
3.2.4 Creating A Stand-Alone Application	32
3.3 C++	33
3.3.1 C++ Programming	33
3.3.2 Borland C++ For Windows Environment	37
3.3.3 The Object Windows Library	39
Chapter 4	41
Actor Applications.....	41
4.1 Introduction	41
4.2 Flexural Capacity of Tee beam.....	42
4.2.1 Overview of the Tee Beam Application.....	42
4.2.2 Class Structure	46
4.3 Design of Simply Supported Beam	49
4.3.1 Overview of the Simply Supported Beam Application	49
4.3.2 Class Structure	58
4.4 Analysis and Design of Reinforced Concrete Continuous Beam	62
4.4.1 Overview of the Continuous Beam Design Application	62
4.4.2 Class Structure	69
Chapter 5	78
C++ Applications.....	78

5.1 Introduction	78
5.2 Flexural Capacity of Tee Beam Application	78
5.3 Analysis and Design of R. C. Continuous Beam.....	83
Chapter 6	90
Results.....	90
6.1 Comparison of Actor and C++	90
6.1.1 Syntax	90
6.1.2 Environments.....	91
6.2 Applicability of Object Oriented Programming in Structural Engineering.....	93
Chapter 7	98
Summary and Conclusions	98
7.1 Summary.....	98
7.2 Conclusions.....	98
References	100
Appendix A	103
Appendix B	115

List of Figures

Figure 3.1 Action Method 'setCaption'	20
Figure 3.2 Method 'sections'.....	21
Figure 3.3 Start of an Actor Session	24
Figure 3.4 Actor Display and Actor Workplace.....	25
Figure 3.5 Actor Class Browser	27
Figure 3.6 Actor Attribute Browser	28
Figure 3.7 Actor Inspector.....	30
Figure 3.8 Attribute Definition for a simple window.....	31
Figure 3.9 Class definition for class 'teeBeam'	34
Figure 3.10 Class definition for class 'teeBeam'	35
Figure 3.11 Member function 'Input'	35
Figure 3.12 Class definitions for 'mainClass' and 'subClass'	37
Figure 3.13 Borland C++ for Windows Integrated Environment.....	38
Figure 4.1 Main Menu: Flexural Capacity of Tee Beam.....	42
Figure 4.2 Flexural Capacity of Tee Beam: Material Properties Dialog.....	43
Figure 4.3 Flexural Capacity of Tee Beam: Invalid data message box	44
Figure 4.4 Flexural Capacity of Tee Beam: Output Screen	45
Figure 4.5 Flexural Capacity of Tee Beam: Section properties dialog box.....	45
Figure 4.6 Class Structure for Tee Beam Application.....	47
Figure 4.7 Design of Simple Beam: Main Window	50
Figure 4.8 Design of Simple Beam: Geometric Properties	50
Figure 4.9 Design of Simple Beam: Material Properties.....	51
Figure 4.10 Design of Simple Beam: Load Cases	51
Figure 4.11 Design of Simple Beam: Distributed Load Dialog Box	52
Figure 4.12 Design of Simple Beam: Partially Distributed Load Dialog Box.....	53

Figure 4.13 Design of Simple Beam: Concentrated Load Dialog Box.....	53
Figure 4.14 Design of Simple Beam: Left Support Moment Dialog Box.....	54
Figure 4.15 Design of Simple Beam: Right Support Moment Dialog Box.....	54
Figure 4.16 Design of Simple Beam: Load Combination Dialog Box.....	55
Figure 4.17 Design of Simple Beam: Recommended Section.....	56
Figure 4.18 Design of Simple Beam: Flexural Steel Dialog Box	57
Figure 4.19 Design of Simple Beam: Stirrup Bar Dialog Box	57
Figure 4.20 Design of Simple Beam: Results.....	58
Figure 4.21 Class Structure for Simple Beam Application	59
Figure 4.22 Flow-Chart of object 'simpObject'.....	61
Figure 4.23 Design of Continuous Beam: Main Window	63
Figure 4.24 Design of Continuous Beam: Geometric Properties	63
Figure 4.25 Design of Continuous Beam: Material Properties Dialog Box.....	64
Figure 4.26 Design of Continuous Beam: Load Combinations.....	65
Figure 4.27 Design of Continuous Beam : Distributed Load.....	66
Figure 4.28 Design of Continuous Beam: Given Section Properties.....	66
Figure 4.29 Design of Continuous Beam: Design Steel Ratio Dialog Box.....	67
Figure 4.30 Design of Continuous Beam: Design Section.....	68
Figure 4.31 Design of Continuous Beam: Results.....	69
Figure 4.32 Class Structure for Continuous Beam Application	70
Figure 4.33 "Geometry.wdl" file	72
Figure 4.34 Method 'nextSpan'.....	73
Figure 4.35 Flow Chart of Object 'Analyze'.....	75
Figure 4.36 Flow Chart of Object 'design'.....	77
Figure 5.1 Class Structure for the Tee Beam Application	80
Figure 5.2 Class Definition for 'TTeeWindow' class.....	81
Figure 5.3 Class Structure for the Continuous Beam Application	84

Figure 5.4 Class definition for the class 'TCaseDlg'..... 86

List of Tables

Table 6.1 Comparison of Actor and C++ 94

**Table B.1 Comparison of Subroutines in Holzer's Frame Analysis Program and Functions
in the 'Analysis' class of the Continuous Beam Application..... 116**

Chapter 1

Introduction

1.1 Introduction

The computer industry has experienced profound advances in computer hardware. Although there have been significant improvements in software technology, these changes have always fallen behind those in hardware with the result that there is, typically a three to five year lag between the introduction of new hardware and the development of software to support the hardware. Until recently, most computer programs were being developed using the structured programming approach developed in the 1970's. However, in the last few years many new approaches have been introduced. These approaches, such as object oriented programming and neural networks are currently being tried and tested.

The object oriented programming (OOP) approach has recently received wide attention. Many of the new programs developed in the last few years are based on the Object Oriented Programming approach. It is anticipated that object oriented programming will revolutionize the software industry and change the way that programmers look at program development. The reasons for this has to do with many advantages that object oriented programming has over conventional procedural programming technique. Some of the advantages include: a) ease of software development, b) ease of maintenance, and c) reusability.

The computerization of the structural analysis and design process has made it necessary for structural engineers to have some knowledge of programming. It has also opened another field for structural engineers: structural engineering software development. However, structural engineering software developers are still using the old structured programming approach. Very few examples could be found in the structural engineering journals of work related to the use of object oriented programming in

structural engineering. The need for this research was motivated by the fact that few attempts have been made by structural engineers to use the object oriented programming approach for the development of structural engineering applications.

1.2 Purpose and Scope

In this study the use of object oriented programming for developing structural engineering applications in the Microsoft Windows graphical user environment is studied. The primary objective of the research work was to study the use of object oriented programming technique for developing structural engineering applications. Two very different object oriented programming languages, Actor and C++, were considered in this study. Actor was selected because it is a pure object oriented programming environment. The selection of C++, which is a hybrid language, was based on the fact that it is currently the most popular object oriented programming language.

A simple program to compute the flexural capacity of reinforced concrete tee beam was developed in both languages in order to develop an understanding of the object oriented techniques involved. After developing the tee beam application in Actor, a program for the design of simply supported reinforced concrete beams was developed. The two languages, Actor and C++ were used to develop a program for the analysis and design of reinforced concrete continuous beams. This application is representative of a typical real world structural engineering application and has both analysis and design components. The procedures and techniques in the development of this application can easily be applied to the development of other structural engineering applications, such as for example, a program for the analysis and design of two or three dimensional trusses and frames.

The scope of the work involved study of object oriented programming techniques for structural engineering applications. During the course of the development, some of the important features of object oriented programming such as encapsulation, abstraction,

inheritance, and reusability were studied. The study presents a comparison of the two languages and discusses some of the advantages and disadvantages of using object oriented programming in structural engineering.

1.3 Organization

An introduction to object oriented programming is given in Chapter 2. A review of the literature on object oriented programming and its application in structural programming is also presented in Chapter 2. Chapter 3 contains a discussion of object oriented programming languages, Actor and C++. Chapter 4 presents a detailed description of the three applications developed in Actor. The class structure and hierarchy is also presented. Chapter 5 describes the two applications developed in C++. Chapter 6 presents a comparison of Actor and C++. This chapter also contains a discussion of the advantages and disadvantages of using object oriented programming in structural engineering. A summary of the study and its major conclusions are given in Chapter 7.

Chapter 2

An Overview of Object Oriented Programming

2.1 Introduction

In this chapter an overview of the basic concepts of object oriented programming is presented. Since object oriented programming is in an evolutionary stage, few structural engineers are fully aware of the basic concepts of object oriented technology. A discussion of existing programming methodologies is presented, followed by a description of the basic concepts of object oriented programming. Some of the advantages and disadvantages of object oriented programming when compared to procedural programming are discussed and the object oriented languages are described. This chapter also discusses the application of object oriented technology in structural engineering and presents a review of some existing structural engineering applications that were developed using this technology.

2.2 Programming Methodologies

A programming paradigm is the method of approaching a programming problem. There are a number of ways in which a programming problem can be handled. These include: conventional modular programming, rule based programming, serial programming and parallel programming. In conventional modular programming, the problem is divided into a number of smaller problems. These small parts are further subdivided into smaller parts. This process is continued until an individual part becomes easily manageable. In rule based programming the program consists of a series of rules which operate on the data. In serial programming, only one arithmetical or logical operation is executed at a time. Conventional procedural programming follows the serial programming model in that a function calls to execute the next function and only one function is executed at any time.

In parallel programming, concurrent operations are performed simultaneously. Although there are many programming techniques, the most commonly used approach is conventional modular programming. So whenever a new programming technique is developed, it is compared with the conventional procedural programming technique.

2.3 Object Oriented Programming

The fundamental idea behind object oriented programming is to combine into a single unit both data and functions (or methods) that operate on the data. In object oriented programming, classes are defined which combine data with functions. Classes are the passive part of object oriented programming and define the behavior of objects. Objects, which are derived from classes, thus have access to both the data and the functions that operate on the data. Object oriented technology reserves computer memory for different objects. This allows each object to act independently of the others. The programmer utilizes this independence to use the object at any location without duplication. This eliminates the possibility of overwriting memory locations.

The partitioning of memory protects the data of the object from being accessed by other objects. The data of an object can only be accessed by the functions that have been defined for that object. Other objects can not access this data although other objects can make a request to a particular object to access its own data and pass it to them. The result of this is that each object is independent and changes in an object do not affect the behavior of other objects in the program. This feature is called as encapsulation of data and it provides increased modularity in object oriented programming. Since object technology revolves around objects, it is necessary for the objects to be able to interact with each other. To facilitate this interaction, functions declared for an object are generally made known to other objects. This interface feature is called object abstraction. It allows the object to make its behavior known to other objects.

A class can be used to create more than one object of that class. Each object acts independently of the others and has its own set of data members. This capability of creating number of objects from a single definition of a class, saves considerable programming effort. Also subclasses can be derived from the base class to add functionality to a class. An object derived from a sub class can be used to provide additional functionality that may not be available in an object derived from the base class. This feature is called inheritance and allows the programmer to derive new classes from an existing class to suit the requirements of the application under development.

Another way of using inheritance is to redefine functions in the base class so that the corresponding function in the inherited class does some other task that is more appropriate for the inherited class. This feature of changing the behavior of a function in an inherited or derived class that has the same function name as the function in the base class is called as polymorphism. Thus, the same function call in a program will result in different behavior of an object depending upon the class of the object.

2.4 Elements of Object Oriented Programming

The key elements of object oriented programming are the following :

2.4.1 Classes and Inheritance

A class is a data structure that combines data elements with functions for manipulating the data elements. In a pure object oriented programming language, like Actor, classes are also objects, but with very limited capabilities. A class defines the behavior of objects. It provides a template from which objects can be created and used. The definition of a class does not allocate memory for its data members. A class also provides a way to hide data which is an essential part of object oriented programming. The class definition controls the access to data and the functions (or methods) defined in

the class. For example, in C++ access is controlled by using the keywords 'private', 'public' and 'protected' as discussed later in Chapter 3.

Another important feature of object oriented programming is inheritance. When a class is defined and provided with data and methods, it can be used to create new classes that are derived from the original or base class. These new classes inherit both the data and the methods that were defined for the base class. Also, new data and methods can be added to the new class. So the inherited class will have the features inherited from its parent class as well as any new features defined for the class. This allows the opportunity to use all of the features provided in the parent class and provides additional functionality as needed. Many object oriented programming languages allow what is known as multiple inheritance. In multiple inheritance, a class can be derived from more than one class. The new class inherits functions of all parent classes. It contains both the data elements and the functions contained in the parent classes. Multiple inheritance is a tool which needs serious thought before its use. It adds to the complexity of the program and the memory storage requirement. Although it is a useful feature, it should be used sparingly. It is usually possible to fine tune an existing class to perform the required task without having to use multiple inheritance.

2.4.2 Object

An object is the most important element of object oriented programming. An object is an independent identity which has its own data as well as functions that can operate on the data. The behavior of an object is based on its class definition. When an object is created from a class, memory is allocated for the data elements of the object. When an object needs to perform a given task it calls the corresponding function to operate upon its own data. For example, if the program needs to print a string, a message is sent to a string object to print itself. The string object has functions that know how to print itself. It also has as one of its data elements the string to be printed. The

programmer just sends a message to the object to perform the desired task and doesn't have to worry about the details of how the task is implemented since these details are taken care of by the object. Thus, an object is an active entity which knows about the data and also knows how to operate upon the data. This technique of encapsulation is a vital element of object oriented technology. In object oriented programming, data and procedures of any object are invisible from outside while the interface to the object is very well defined. This ensures that if changes are required in the behavior of any object, the changes can be made internally without changing the interface.

2.4.3 Message Passing

Message passing is similar to calling a function in a procedural language. An object communicates with other objects by sending messages. The body of the message contains the name of the message, name of the object to which it is passed and any arguments such as variables, known to the message sending object and which the server object might need. Generally an object accepts messages it knows and ignores all other messages. This feature is used further in C++ where depending upon the number of arguments in a message, the message goes to a different object which makes it possible to have the same message name but with a different number of arguments. This is called operator overloading. Message passing is very important in object oriented technology. It preserves the autonomy of each object.

2.5 Object Oriented Programming : Why is it needed ?

The need for object oriented programming is best explained by examining the shortcomings of conventional procedural programming and the corresponding advantages of object oriented programming. Procedural programs contain data and instructions to the computer for operating on the data. When the problem size becomes large, this list of

instructions is broken down into small modules. This makes program development easier and reduces the complexity of the problem. Dividing a program into functions and modules is one of the essential elements of structured programming. However, as the program size increases, using the procedural programming approach becomes increasingly difficult. One of the basic disadvantages of the procedural programming paradigm for large complex programs ^{has to do with} ~~is~~ the role played by the data [25]. In procedural programming, the primary emphasis is on functions rather than on the data even though these functions act on the data.

In procedural programming, functions and data are treated as separate entities. The functions perform various operations on data. If many functions need access to the same data then the data are stored as global variables. The use of global variables greatly increases the chances of the data being accidentally corrupted by a function. Another problem is that, when the data structure is modified, for example, if new data items are added, it becomes necessary to modify all of the functions that access the data. This can be very troublesome in a large program with many functions. Since in most large programs a function has to access data that are outside the body of the function, the function is not self sufficient. When writing new functions it is easy to corrupt data in other parts of the program. Another problem associated with structured programming is the lack of relationship between the elements of the program and the real world. When studying a procedural program it is not always obvious what the program modules and the data represent. It is also difficult to determine what functions are needed for program development unless the programmer actually begins to write some functions which then provide clues for the next functions [29].

One more short coming of procedural programming is the inability to support new data types. Many conventional programming languages do not have provisions for defining new data types. These languages provide several fixed data types such as integer, real or float, and character but do not have the capability for defining new data types such

as, for example, data type for representing dates or points with x, y, z co-ordinates. In a conventional programming language, that does not support derived data types, it becomes necessary to handle each element of each new data type separately. With object oriented programming it is very easy to create new data types.

One of the most important feature of object oriented programming is that in an object oriented programming language both the data and the functions that operate on the data are combined in one object. Thus, every object has its own data and functions. If an outside function wants to access the data of another object it has to make a request to the object of that class to access the data. In this way the integrity of each object is maintained and the data is secured. Also, if at a later date it becomes necessary to alter the data structure it is only necessary to make sure that only the functions that are a part of the object itself work with the new data. It isn't necessary to modify functions in other objects. This enhances program maintainability [25].

The integrity of each object makes it easier for functions to be written separately. Thus, a large problem can be divided into smaller parts, with each part having its own data and functions. This makes it easier for different programmers to work on different parts of a large project independently and makes software development very efficient. Another advantage of object technology is code reusability. Once a class is created and completely tested it can be reused in several places. The feature of inheritance (Section 2.4.1) allows the programmer to define new classes from previously tested classes [7].

In object oriented programming, there is a hierarchy of classes that looks similar to the real world since there are parallels between objects in object oriented programming and objects in the real world. Thus, object oriented programs resemble real world situations more closely and are easier to understand. For example, a program that does frame analysis, would have objects of loads, and members which would prototype objects in the real structure [7].

Another important asset of object oriented programming is extensibility. New data types and the functions that operate on these data can easily be created. This makes it possible to enhance the capabilities of the language. For example, to create a new data type called 'date', all that is necessary is to create a class called 'date' with data elements of 'day', 'month', and 'year' and functions (or methods) to read the date, store the date and any additional functions, such as, a function to determine the number of days between any two dates. Once this is done it is possible to define any number of objects of the class 'date', each with its own data elements for day, month and year [25].

2.6 Object Oriented Programming Languages

There are now a number of object oriented programming languages. Examples of these languages include C++, ACTOR, Smalltalk, Trellis, Objective C, Object Pascal and Flavors. Of these only a handful are enjoying commercial success. The three most successful object oriented programming languages are C++, Objective C and Smalltalk [29].

Smalltalk is the oldest object oriented language and it has come a long way from its ancestor, Simula. It is a "pure" object oriented programming language as opposed to C++ or Objective C which are "hybrid" object oriented programming languages. Both C++ and Objective C are extensions to the conventional C programming language. C++ was developed at Bell Labs by AT & T and is rapidly emerging as the standard for object oriented programming languages. Objective C was used for developing the object oriented programming applications in the NextStep operating system. Digital Equipment Corporation has its own object oriented programming language called Trellis [2].

Object oriented programming languages can be divided into two categories :

1. Pure object oriented programming languages,
2. Hybrid object oriented programming languages.

In a pure object oriented programming language all programs contain objects, class messages, and functions. Even the class 'class' is an object from which subclasses can be derived. Examples of pure object oriented programming languages are Smalltalk and ACTOR. Since everything is object in these languages whatever is done has to follow the ground rules of object technology [30].

Hybrid languages are extensions to conventional procedural programming languages. Examples of hybrid object oriented languages are C++, Objective C, and Object Pascal. Since these languages are extensions to conventional procedural programming languages, many programmers, who are familiar with conventional programming language, prefer hybrid object oriented programming languages to pure object oriented programming languages. The hybrid languages give the programmer the freedom to use as little or as much of the object oriented programming feature of the language as needed. It also allows programmer to make use of previously written code. This increased flexibility is an important reason for the success of the hybrid object oriented programming languages [2].

The two object oriented programming languages that are currently dominating the market are C++ and Smalltalk. C++ is a widely accepted object oriented programming language. The reason for its popularity is primarily due to the fact that it is an extension of the C programming language which has a very wide market. Among the pure object oriented programming languages, Smalltalk is the most successful. Actor and Eiffel have minor market shares [29].

2.7 Shortcomings of Object Oriented Programming

While object oriented programming has many advantages it also has a few disadvantages. One of these is the additional responsibility that is placed on the programmer. Although object technology provides increased program modularity, a

considerable amount of additional effort is required in planning the program modules. It is also necessary to have a good understanding of the problem domain. The entire software development project has to be carefully planned in advance to get the maximum benefit of the modularity provided by object oriented programming [12].

To take advantage of reusability of objects it is important to keep objects and classes as generic as possible. Most of the object oriented programming languages provide class libraries and it is important to become thoroughly familiar with these class libraries in order to reduce the unnecessary creation of classes. Although using object oriented programming technology can be somewhat demanding initially, with experience the programmer can take advantage of the many predefined objects provided in the class library and can significantly reduce the time and effort required in developing an application [12].

Another disadvantage is the overhead associated with object oriented programming. Object oriented programs require more resources in terms of memory and execution time when compared to conventional procedural programming. This is partly due to message passing and runtime binding. Runtime binding (also called late or dynamic binding) refers to the execution of a particular function depending upon the class of the object. However this binding time can be reduced by type casting. Although this shortcoming is legitimate, it is not very important. The object oriented approach generally results in a better user interface. The increase in the user efficiency results from a user friendly interface and this compensates for any inefficiencies in computations resulting from the use of object oriented technology [12].

Thus, to summarize it can be said that the shortcomings of object oriented programming are outweighed by the advantages associated with it. Already a large proportion of the software industry is using object oriented programming for the new projects.

2.8 Literature Review

In this section a brief review of the literature on object oriented programming is presented. The review begins with some studies about object oriented programming initially used to understand the concepts of object oriented programming. This is followed by a review of published literature on the application of object oriented programming in civil engineering and structural engineering.

E. H. Tyugu [1] described the basic concepts of object oriented programming. The article covered the classes, objects, methods and the message passing in object oriented programming. The article provided a basic ground for understanding object programming and also illustrated the possibility of concurrent object oriented programming.

A comparison between different object oriented programming languages was the topic of discussion in a paper by W. Schubert and H. Jungklaussen [2]. The discussion focused on the semblance of programming languages to the human thoughts and dialects. The object oriented programming languages were discussed and their relationship with the natural languages was studied. A similar study of object oriented programming languages was done by J. Micallef [3]. The study covered C++, Objective C, Smalltalk, and Trellis. A comparison of these languages with respect to the main criteria of object technology, encapsulation, reusability and extensibility, was presented. The feature of extensibility in object oriented programming was compared with the type abstraction technique by William R. Cook [4]. The study focused on the difference between data abstraction achieved through object oriented programming and conventional programming. The abstract data types depends on the type abstraction while object technology supports it by procedural abstraction. According to Cook, object programming supports both techniques. J. W. Hopkins [5] presented a comparison of Objective C and C++. The paper contained an introduction to object oriented programming and then went on to compare Objective C and C++.

The most important advantage of object oriented programming is the increased maintainability of the code. John Lewis [6] studied the increased maintainability feature of object technology. The study found that object technology provided significantly improved maintainability of a large program compared to the same program developed in a conventional programming style. The study based its conclusions on the maintenance time, error counts and programmer's impression.

The review of current journal papers dealing with object technology in structural engineering can be best started with G. R. Miller [7]. He studied the use of object technology for structural engineering applications. Miller illustrated the basics of object oriented programming. Miller also discussed the object oriented technology and the structural engineering applications. He also realized the limitations of object oriented programming such as the complexity of reckoning the objects and their relationship. The study acknowledged a definite shift towards object oriented programming for structural engineering application development. To prove his point Miller developed a LISP based data base management study [DBMS] [9][10], using object oriented concepts. The articles covered two example programs developed using LISP. The first example described a general matrix inversion algorithm using object technology. The second example is of a frame analysis program for incrementally developing the frame model. Miller also addressed the issue of concurrent programming in structural engineering through object oriented programming [11]. He used Common LISP language to develop a structural analysis program. He created a object database and used the presence of multiple objects from this database to attain limited concurrency in the analysis.

Watson and Chan [8] developed a PROLOG based database management system [PBASE], using an object oriented extension of PROLOG. The DBMS was developed for use in engineering application with special consideration to structural engineering applications. PBASE supports design evolution and allows changes in the data during the evolution.

G. H. Powell developed structural engineering database systems and frame analysis programs using object technology [12]. Powell presented the concepts of object oriented programming and shortcomings of object oriented technology for engineering databases. Powell inferred that while the planning involved in object oriented design of data bases was quite demanding, encapsulation and inheritance made it a better choice for engineering programs. One of the articles by Powell et. al.[13] described the use of object oriented technology for developing a data base system for structural modeling of frames. He proposed an object oriented data base because it proved to be the best solution for the needs of structural engineering. Powell et. al. [21] also addressed the use of object oriented programming in the management of a central database for a structural engineering design program. The study basically dealt with the interaction between the application program and the central database. A database management system using objects was presented as an interface between the application program and the central database. Powell et. al.[14] also used an object oriented programming algorithm for stiffness modeling of a frame structure. He described the concepts of object technology and then illustrated the algorithm. The algorithm can be used to create the frame model from a data base, create the stiffness model, and reverse the node model into connections and components while presenting the results. Powell et. al.[15] developed a reinforced concrete frame design program. The object oriented extension to C, C++, was used with the InterView library of classes. Another example of structural engineering application that uses object oriented programming is a program developed by H. Adeli and George Yu [16] to design a steel girder. C++ was used for the program development. The program had individual objects for the design of the various components of the girder such as the flange, web, and welds. Garrett [20] used object oriented technology to create a database for design standards. He used encapsulation in the objects to test if a particular check in the design standards is satisfied or not. Every object had the value to be checked, as its data, and the code requirements, as its functions which operated upon the data. This

way each object had one of the three results : check "satisfied" or "violated" or "not applicable".

Several other studies have been conducted on the use of object technology in civil engineering. Philippe Remy et. al. [17] used object oriented philosophy to develop a fully interactive user interface for a finite element program. The program can be run on Macintosh, MS-Windows, and OSF/Motif. Michael Rice developed a freeway analysis and design program using object oriented programming [18]. Another attempt at modeling civil engineering applications was done by Amir A. Oloufa [19]. The program modeled a customer-server scheme for an earth moving project using a new object oriented programming language MODSIM. The program was compared with a conventional program called SIAMAN.

In this chapter an overview of object oriented programming was presented. A brief discussion of programming methodologies was presented in the beginning followed by a discussion of classes, methods and objects. The advantages of object oriented programming were discussed followed by a discussion of object programming languages. The shortcomings of the object oriented programming were also illustrated. The last section in the chapter presented a brief literature review. In the next chapter the object oriented programming languages Actor and C++ are discussed.

Chapter 3

Overview of Actor and C++

3.1 Introduction

This chapter describes the two object-oriented programming languages considered in this study - Actor and C++. Actor is a pure object oriented language while C++ is a hybrid object oriented language. The main features of the two languages and some of the main differences between them are discussed. Additional information about the syntax of programming is available in the programming manuals [Ref. 30, 31] and related books [Ref. 22-26, 28, 29, 32].

The Actor programming language is discussed first, followed by a description of the Actor environment. Since Actor is an extensible language, the process of creating an executable file in Actor is somewhat different from the usual process of compiling and linking. This is discussed in the section on creating a stand-alone application. The discussion on C++ focuses primarily on the Borland C++ compiler. First, a brief description of the important terms associated with C++ is presented. Then the Borland C++ environment for Windows is described. The chapter concludes with a brief description of the Borland C++ Object Windows Library which was used for developing the C++ Windows applications.

3.2 Actor

Actor, developed by WhiteWater Group, is one of the few pure object oriented programming environments for developing Microsoft Windows applications. In Actor, classes are also objects. The topmost class 'Object' in Actor, serves as the root class. All other classes, either provided with Actor or developed by the user, are derived from 'Object' either directly or through a chain of inheritance. Although the class 'Object' is the meta class, it has very few capabilities.

Actor is an extensible environment. This means the application program developed by the user becomes part of the Actor class library. The advantage of this is that user defined classes are easily available for future applications. The availability of user defined classes from within Actor allows the programmer to take maximum advantage of the inheritance feature of object oriented programming.

3.2.1 Actor Programming

In this section a brief description of the basics of Actor programming is presented. Since Actor is a pure object oriented programming language, the programming involves developing classes that can do specific tasks. A brief description of classes and objects is presented, followed by a description of methods. Since variables are the data members of the objects on which the methods act, they are presented at the end.

Classes and Objects

All Actor applications contain classes that provide definition of objects. Classes define the data variables and the methods and are the basic framework from which objects can be developed. The objects which know the data and the behavior are the tools that actually do the work in an application. It is the creation of an object and not the definition of a class that sets aside computer memory. When developing application programs in Actor, objects of server class have to be created.

In order to create an object, a creation message is passed to the class. This causes the class to create an object of itself. This object has the behavior defined by the class and contains all the data elements defined for the class. When an object is created, memory is allocated for the object. An example of an object creation message is the following.

```
trial := new(design);
```

Here a new object 'trial' is created from the class 'design'. The object 'trial' can then be operated upon to do the necessary design. Some classes, like the class 'Array', accept an argument for the object definition. For example, the statement

```
trial := new(Array,10);
```

creates an object 'trial' with ten elements of the class Array.

Number classes such as Real and Integer do not need a formal object definition. Assigning a number or a string value to a variable automatically assigns the variable as an object of the particular class. For example, in the statement

```
a := 3;
```

the object 'a' will be an object of class Integer having a value of 3. Here the class of the object 'a' is not fixed till the value '3' is assigned to it. So 'a' can be assigned to any class such as a real 3.0, an integer 3, or a string "3.0".

Methods

Methods in Actor are somewhat similar to functions in C. Methods define the behavior of the objects. Methods operate upon data, perform computations, transfer information to and from other objects and can also transfer control to another methods. A method can also be written to return a value. This returned value can be used to determine the subsequent flow of the program.

There are two types of methods in Actor : Action Method and Method. An Action method is executed as a consequence of an event. The event can be the click of a mouse button, selection of a button, selection of a menu item or simply the creation of a window. The Action method specifies the action which will cause execution of the method. For example, consider the method 'setCaption' shown in Figure 3.1.

```
Action setCaption (self,msg) #[#created]
{
    setTitle(self,"Design of Continuous Beam");
}
```

Figure 3.1 Action Method 'setCaption'

In the example shown in Figure 3.1, the 'Action' keyword in the first statement defines the method 'setCaption' as an Action method. The method will be executed each time an object of the class is 'created'. The list in parentheses following the name of the method contains two variables: the object name, in this case the same object (called self), and the message. The parameters in parentheses can be used to pass arguments to methods. The argument passing technique can be used only for methods but not for Action methods. Since all methods have access to instance variables and class variables, it is not always necessary to pass variables as arguments.

A 'Method' performs a task when asked to, by any other method or object. It is similar to a sub-routine in a procedural language. Since methods are defined for a class they have access to class data. This ensures encapsulation of data within objects. An example of a method is given in Figure 3.2.

```

Def sections(self,k|j)
{
/***** IMPORTANT VARIABLES *****/
    x = distance of the section from the left support
    k = span number
    j = counter
*****/
x := new (Array, 31); /* Defining x as an object of 'Array'
                      with 31 elements */
j := 0;
loop
while j < 31
begin
    x[j] := span[k] / 30.0 * j ;
    j := j + 1;
endLoop;
}

```

Figure 3.2 Method 'sections'

In the method shown in Figure 3.2, the first statement defines the method 'sections'. The variable 'k' is passed to the method as an argument. This variable 'k' represents the span number in the continuous beam design program. The variable 'j', which is used as a local

variable is also defined in the same statement. As the method is part of the class 'design', used in the continuous beam design program, it has access to the instance variable 'span'. The code for the entire method is enclosed in braces.

Variables

All Actor applications contain variables of different types. These include local variables, instance variables, class variables and global variables. Local variables are local to the method and can not be accessed from any other method. Local variables exist for the duration of time that the method is executing and are destroyed immediately after the method has completed execution.

Instance variables are the most important variables in an Actor application. Instance variables are associated with objects. Each object of a class, has its own set of instance variables. This also means that an object's instance variables can be accessed only by that object. This ensures encapsulation of the data and procedures to its highest level. If an object needs to access another object's data it has to make a request to the object to pass the data. This can be accomplished by writing methods that return data.

Global variables can be accessed by any object of any class in the program. These variables offer the least security against data corruption, and hence should be used sparingly. Global variables in an Actor program are used in the initialization of the main application window. A class variable is accessible to all objects of a class or its sub class but are not accessible to objects of other classes. Thus, in this case all objects of a class share the same set of variables defined for the class. Class variables can be useful when all objects share the same data. It is common practice in Actor programming to use a '\$' character as the first letter of the variable name. For example, \$Ast, is a class variable that represents the area of tension steel.

Of the above four variable types, Instance variables and local variables are the most effective for preserving object modularity and for data hiding. Hence when defining a class it is very important to differentiate between instance variables and local variables.

3.2.2 The Actor Environment

Actor is an application that runs under Microsoft Windows. When the first Actor session is started it opens with two basic windows the 'Actor Workspace' and the 'Actor Display' (see Figure 3.3). The Actor Workspace is like the command line in DOS while the Actor Display is an information window. The menus in the Workspace (see Figure 3.4) can be used to load a source file, inspect a class or create a new object or execute a command. The Actor Workspace can also be used to start the main window of an application under development. This makes the development and testing of the application a simultaneous process. Any 'method' defined for a class is compiled and saved immediately and hence is available for testing. The Actor Display window displays error messages while commands from the workspace are being executed. The Display window also shows the amount of memory utilized to compile a class. The Display also has a very important menu called 'Seal Off' which is used for creating stand-alone applications. This is described later in this chapter.

The Actor Workspace and the Actor Display windows are just the two basic windows in the Actor environment. Although the Actor Workspace executes commands it is not the tool for writing application programs. This is accomplished by the 'Browser'. A Browser window can be opened to define new classes, or to modify an existing class. When a Browser window is opened it contains a list of all available classes with a list of their methods. By default all classes are listed in a hierarchical manner making it easy to see sub classes and inherited methods. Although this can be changed, for example, as the classes can be displayed in alphabetical order, it is not useful to do so.

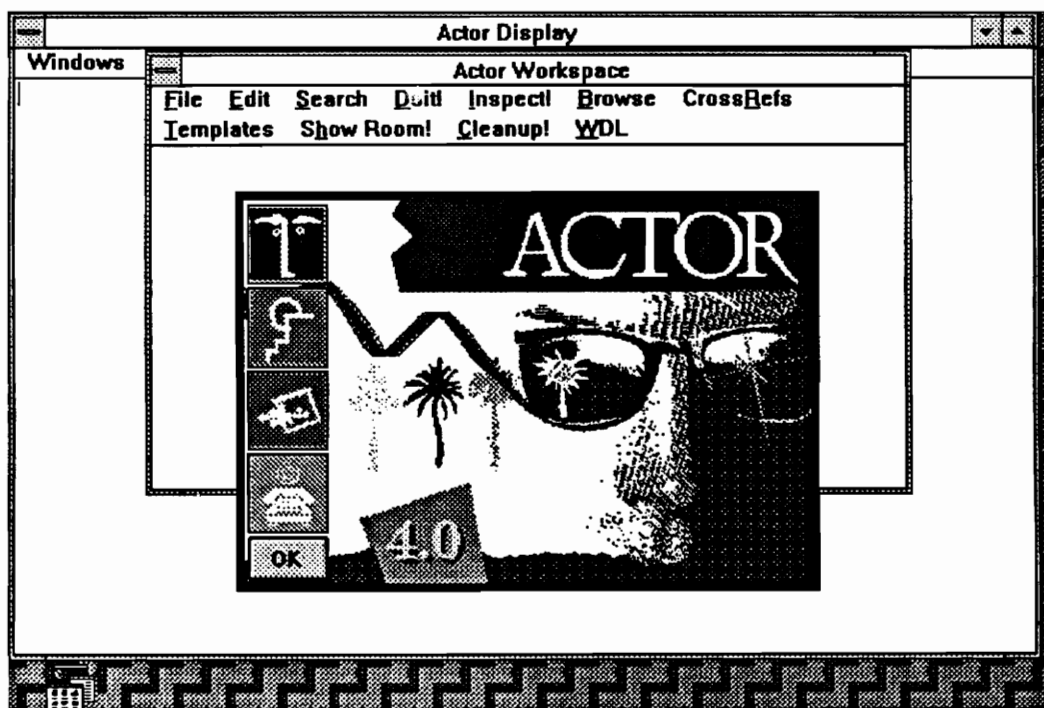


Figure 3.3 Start of an Actor Session

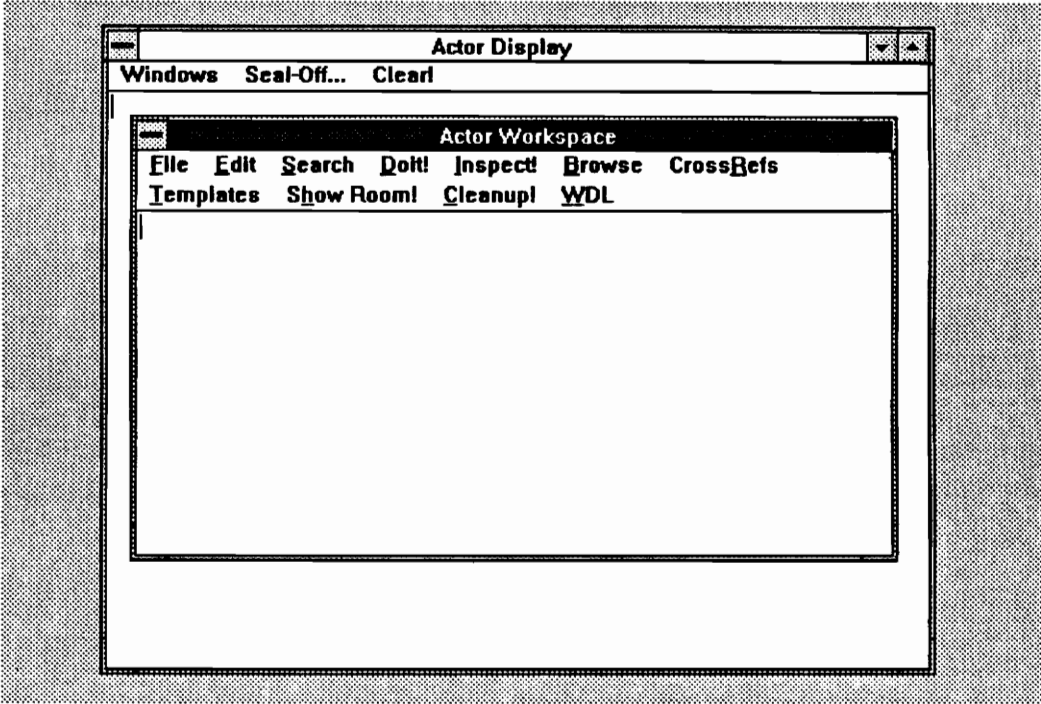


Figure 3.4 Actor Display and Actor Workplace

When an Actor session is started for the first time, all classes in the Browser are the original classes that were provided with Actor. These classes are of a very general nature and provide methods for performing many of the basic operations needed for creating windows applications, such as window creation methods, printer methods, graphics methods, and file management methods. Most new applications make use of these classes. The class list increases as new classes are defined for the application.

In the Class Browser window (see Figure 3.5), a sub class can be defined by selecting the "Make Descendant " sub menu of the "Class" menu. The name of the class is defined in the dialog box. The instance variables are also defined in the same dialog box. A sibling class, parallel in the class structure, can be made in the same way by selecting the "Make Sibling" sub menu. The Browser also lists all the methods defined for each class. By selecting a class from the list of classes, the methods defined for that class appear in the list box. Any method can be viewed and edited by simply selecting it.

Another type of Browser is the Attribute Browser (see Figure 3.6). The Attribute Browser contains the attribute definitions of the dialog boxes used in the program. The attribute definition specifies the creation attributes such as the style and position of the dialog box. The attribute browser also allows the programmer to correlate the resources defined in the resource files to the application. The attribute definition assigns a unique ID number or name to each control in the dialog boxes. These ID numbers or names can be used in the application methods to refer to each control. The attribute definition can also be used to define any data for a control. This feature is useful for assigning initial values to a control before the dialog is displayed.

The Actor Workspace, Actor Display, Actor Browser, and the Attribute Browser are the basic features of the Actor Development environment. Another important tool is the 'Actor Debugger'. The debugger is a valuable tool for debugging applications. As the program grows in size, it becomes very difficult to trace small mistakes. The Actor Debugger gives a list of executed methods up to the method which caused the error. It is

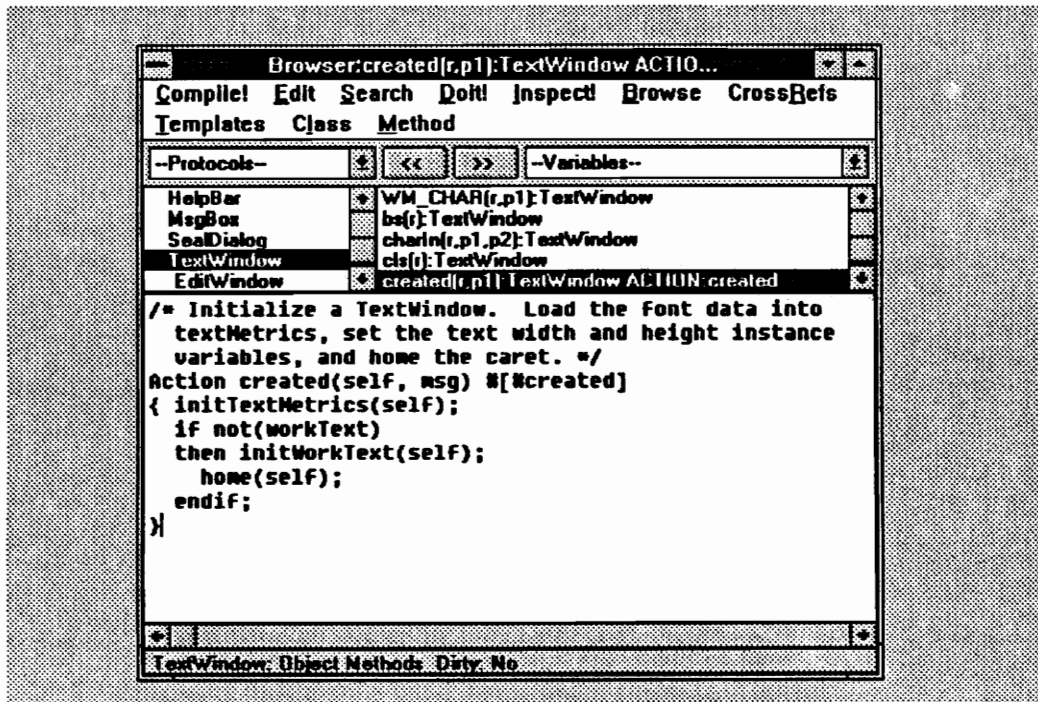


Fig. 3.5 Actor Class Browser

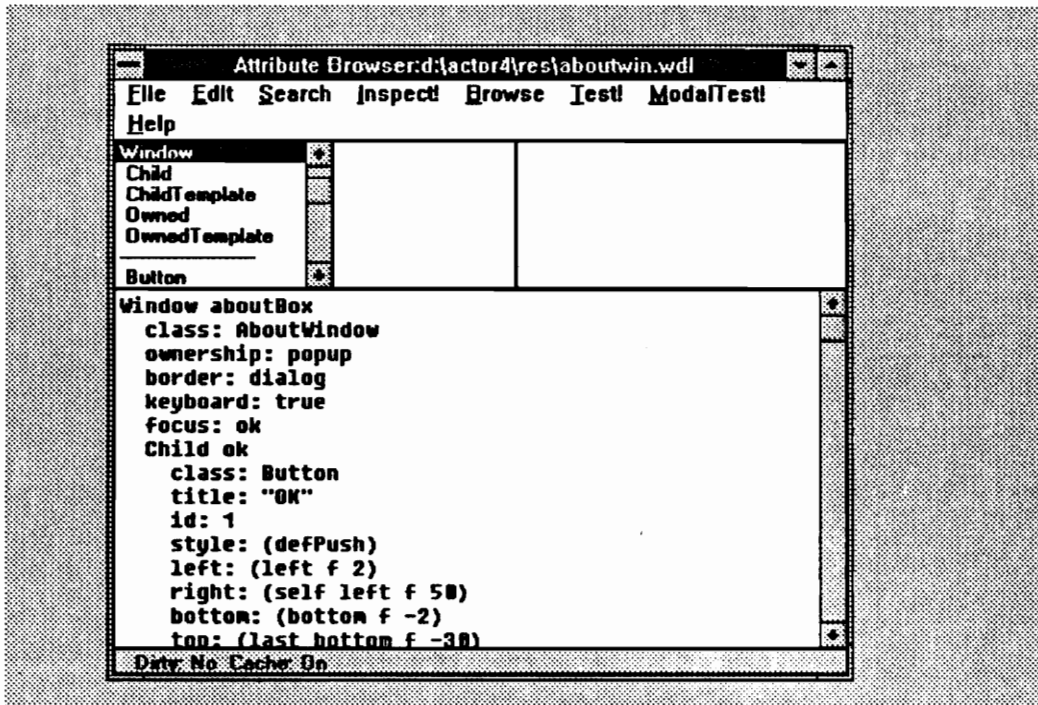


Fig. 3.6 Actor Attribute Browser

possible to move backwards down the list of methods to trace the source of the error. The Actor Debugger, also has a list box which displays the names of variables and the values of local variables for each method at the time of the error. This tool makes backward tracing very easy. Once the error is found and corrected, the application can be resumed from the method that caused the error. This can save considerable time that otherwise would be wasted in having to run the application from the beginning.

The description of the Actor environment would be incomplete without mentioning the 'Inspector' (see Figure 3.7). The Inspector provides information regarding the current status of an object at any time. The Inspector shows the instance variables of an object in a list box. Selecting one of the instance variables makes that instance variable active and its value is displayed in the main window of the Inspector. Since Actor is a pure object oriented programming language, it is possible to view a class in the Inspector. The Inspector shows the class variables and the instance variables associated with the class in the same way. The values of these variables can be checked. This can be very useful when debugging classes.

3.2.3 The WhiteWater Resource Toolkit

All Windows programs make use of resources. Resources include menus, dialog boxes, icons, bitmaps and acceleration tables. Resources are stored in an executable file when the file is linked. The resources can be modified and again included in the executable application. The WhiteWater Resource Toolkit is not an integral part of Actor. However, it is a valuable tool for creating the resources required by an application. The toolkit provides an interactive approach for creating resources. Dialog boxes, icons, menus, bitmaps and other resources can be developed and tested in the toolkit prior to their inclusion in the application. The resource toolkit provides a series of tools for

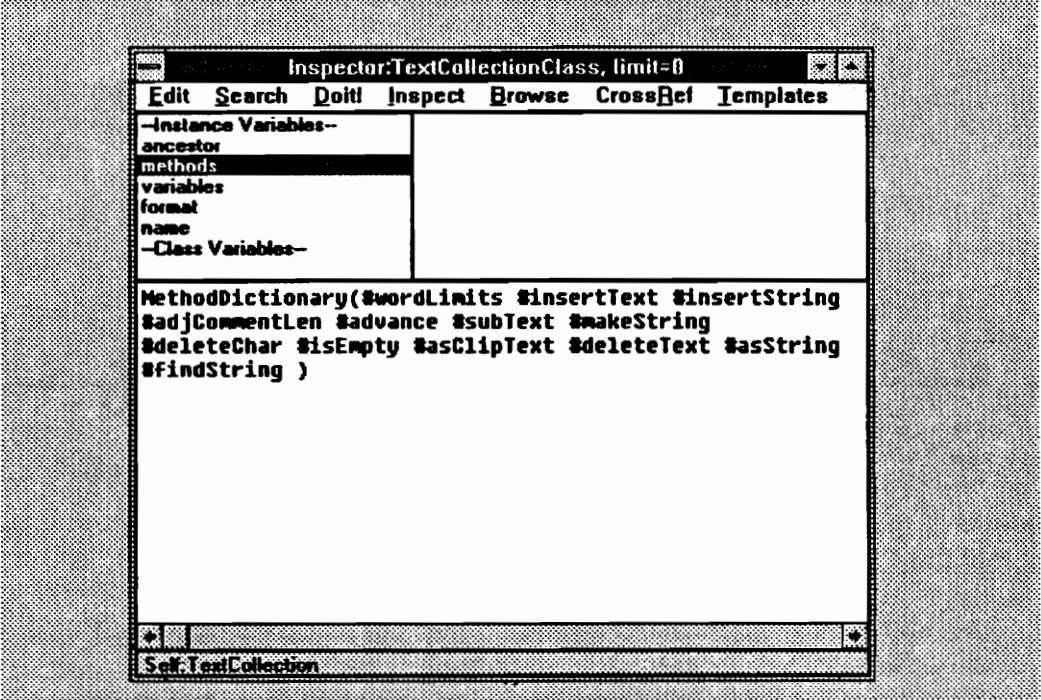


Figure 3.7 Actor Inspector

creating resources. It also has tools for creating various controls such as edit boxes, radio buttons, push buttons, and static text. An important advantage of keeping resources separate from the program code is that this makes it possible to modify these resources without having to rewrite the entire application. For example, it is possible to create two versions of the same application in two different languages such as English and French, by simply creating the resources.

Each dialog box, and the control within a dialog box has a unique ID. This ID allows the application program to access the control. In the program code it is the ID that enables the programmer to reach individual controls of the dialog. In Actor, dialog boxes are linked with the resource definition by using the Windows Description Language (WDL). The WDL file sets the creation attributes for all windows used in the application. This includes the dialog boxes and the main window.

In the example shown in Figure 3.8, an attribute definition for a simple window is illustrated.

```
Window Trial
class : Window
Resource : "Trial_Resource"

Child Child1
class : Button
id : 101
```

Figure 3.8 Attribute Definition for a simple window

In the Figure 3.8 the first statement assigns a name to the dialog box defined in the resource 'Trial_Resource'. The 'class' statement assigns the class 'Window' to the Window 'Trial'. This allows 'Trial' to respond to the methods defined for the class 'Window'. The 'Child' statement defines the Child window of the control class 'Button'. The position and style for the 'Child1' button are defined in the resource file with an ID of 101. This button can be accessed in the source code by referring to its ID or to the name 'Child1'.

3.2.4 Creating A Stand-Alone Application

In most programming languages, an application can not be tested until an executable file is created. However, Actor allows testing of an application during its development. This makes testing of the program much easier. However, the application must be tested within the Actor environment. After the program has been developed and tested, several additional steps have to be performed in order to create a stand-alone executable application.

Since Actor is an extensible programming environment, all user defined classes and methods become part of the Actor Class library. Thus, when the application classes are completely debugged they become part of the library. The procedure of 'Sealing Off', as is called in Actor, removes extra classes from the application. The sealing off procedure also deletes global variables and attaches them to the application.

To start the sealing off process, a new sub class of the 'Application' class is created. This Application class handles the Main Window of the application. It is necessary to create an application class for each application. This class generally has only one method 'initMainWindow' which opens the main window of the application. The sealing off procedure creates an image file, having an extension of .ima, containing the necessary part of the Actor environment. Also, Dynamic and Static memory requirements for the application can be set at the time of sealing off.

Once an image file is created, the application can be tested in the Actor environment with the application image file. This provides an opportunity to check for unexpected errors in the application such as reference to global variables instead of instance variables. Although an image file can be used to run an application independently of the Actor environment it uses Actor's executable file. This means that the image file can't be run without the executable Actor file. An executable file of the application would run in any Windows environment as a stand alone application. This application executable

file will contain only those resources that are needed for the application. The application's resources can be compiled using the Microsoft Resource Compiler and the compiled resource file can then be linked to the application's executable file. This completes the procedure of creating the stand alone application. It should be noted that both the executable file, 'application.exe' and the image file, 'application.ima' are necessary for running the stand alone application in the Windows environment.

3.3 C++

C++ is an extension to the popular C programming language and is the most popular object oriented programming language. The extensions to C provide support for object oriented programming. Since the language and the syntax of C++ is similar to C, it is very popular among C programmers. C++ allows the use of previously developed C functions and it is possible to write a C++ program that does not use any of the object oriented programming features of the language. This provides C programmers the opportunity to use as much, (or as little) of the object oriented programming features and enables a gradual transition from C to C++. In the sections that follow, a brief description of the object-oriented features of the C++ language is presented.

3.3.1 C++ Programming

Class Definition

A class definition provides the template for an object of the class. In C++ the member variables and member functions are declared in the class definition. It should be noted here that, all the class member variables and class member functions must be declared in the class definition. The member functions can be defined in the class definition or can be defined separately outside the class definition. A class definition for the class 'teeBeam' with the necessary data members and functions may look like the one shown in Figure 3.9.

```

class teeBeam
{
    protected :
        float fc,fy,b,bw,Ast,d,phiMn;
    public :
        teeBeam : teeBeam();
        void Input();
        void showOut();
        int ductCheck();
        void mmtCapacity();
};

```

Figure 3.9 Class definition for class 'teeBeam'

The first statement in Figure 3.9 defines the name of the class 'teeBeam'. The data and function definition for the class is enclosed in braces{ }. The keyword 'protected' and 'public' define the subsequent data and functions as protected and public respectively. The semicolon after the closing curly brace is necessary to mark completion of the class definition. The class teeBeam contains protected data variables such as fc, fy, b, bw, Ast, d, phiMn. All these variables are declared to be of type float. The class also contains public functions such as Input(), showOut(), ductCheck(), and mmtCapacity().

An object of class 'teeBeam' is created by the following statement:

```
teeBeam beam1;    // Object beam1 created of class teeBeam
```

After the above statement is executed an object 'beam1' of class 'teeBeam' is created. Also memory is allocated for the data elements of the class when the object is created. Each object of the class 'teeBeam' will have its own copy of the instance variables declared in the class definition such as the variables fc, fy, b, bw, Ast, d and phiMn.

Member Functions

The member functions of a class define the behavior of objects of the class and operate upon data. While the 'main' function of a program defines the general flow of the program, the class functions shape the interaction of objects. Class functions are similar to sub-routines in conventional programs.

A class function can be defined either as private, public or protected. The class functions can be declared and defined in the class definition or can be defined separately. The following example shows how a class function can be declared and defined in the class definition.

```
class teeBeam
{
    protected :
        float fc,fy,b,bw,Ast,d,phiMn;
    public :
        teeBeam : teeBeam() { fc=fy=b=d=Ast=phiMn= 0;};
        void Input();
        void showOut();
        int ductCheck();
        void mmtCapacity();
};
```

Figure 3.10 Class definition for class 'teeBeam'

Here the function 'teeBeam' is declared and defined in the same line. The function that has the same name as name of the class is called the 'constructor'. The constructor is typically used to initialize class variables. The constructor is called each time an object of the class is created.

The function 'Input' is declared but not defined in the class definition. The function can then be defined later as shown below:

```
void teeBeam :: Input()
{
    Input inputDialog;        // Declare object for input dialog
    inputDialog -> ExecDialog(this,"input_dlg");
}
```

Figure 3.11 Member function 'Input'

In the example shown in Figure 3.11, the function 'Input', as defined in the class 'teeBeam', does not have a return value. The function calls a function of the class 'Input' to execute

an input dialog box. The function is declared to be of class 'teeBeam' in the first statement:

```
void teeBeam :: Input()
```

This method of defining classes and functions can be continued to complete the application development.

Public, Private and Protected Variables and Functions

The class variables and functions defined in a class definition can be Public, Private or Protected. Public variables and functions are visible to any function of any class from anywhere in the program. Since in object oriented programming the emphasis is on data hiding, Public variables are generally avoided. However, functions are usually defined as Public so that the objects can be operated upon from any part of the program. Thus, public functions facilitate object abstraction while private data ensure data hiding. Private variables and functions are accessible from the methods declared for that particular class. This is the best method of hiding the data. However, this method also makes inheritance difficult, since sub classes can not access private functions and data variables of the parent class. Protected variables and functions offer greater functionality along with data hiding of the private methods and functions. Protected variables and functions defined for a class are also accessible to derived classes. Thus, a derived class inherits all the protected functions and variables defined for the base class. To illustrate the types of variables consider the class definition shown in Figure 3.12.

In the example shown in Figure 3.12, there are two classes, 'mainClass' and 'subClass'. The class 'subClass' is derived from the 'mainClass'. The definition of 'mainClass' contains a private data member called 'privateData' of type integer and a protected data member 'protectedData'. The private data 'privateData' is not available to the sub class 'subClass' but 'subClass' is able to access the protected data 'protectedData'. The public functions 'addData()', defined in the 'mainClass' and 'addX()', defined in the 'subClass' are accessible from any part of the program.

```

class mainClass
{
private :
    int privateData;
protected :
    int protectedData;
public :
    void addData();
};
class subClass : public mainClass
{
private :
    int x;
public :
    void addX();
};

```

Figure 3.12 Class definitions for 'mainClass' and 'subClass'

3.3.2 Borland C++ For Windows Environment

The applications in this study were developed using the Borland C++ compiler and the Borland C++ Object Windows Library (OWL). Borland C++ for Windows is an Integrated Development Environment (IDE) and consists of a text editor, compiler, linker, make utility and a debugger all under one window (see Figure 3.13). This makes it possible to perform all application development tasks from editing the source code to creating the executable program from within the same application.

In Borland C++ for Windows, a new project is started by selecting the menu 'Open Project'. A typical project in Borland C++ consists of C++ source code files, a module definition file and a resource file. The Project menu has menu choices for adding these files in a project. For large projects more than one C++ source code file can also be added. The module definition file specifies information about the application's code and data segments, the size of the local heap, and the size of the application's program stack. The resource file defines the resources such as dialog boxes, menus, bitmaps, icons and

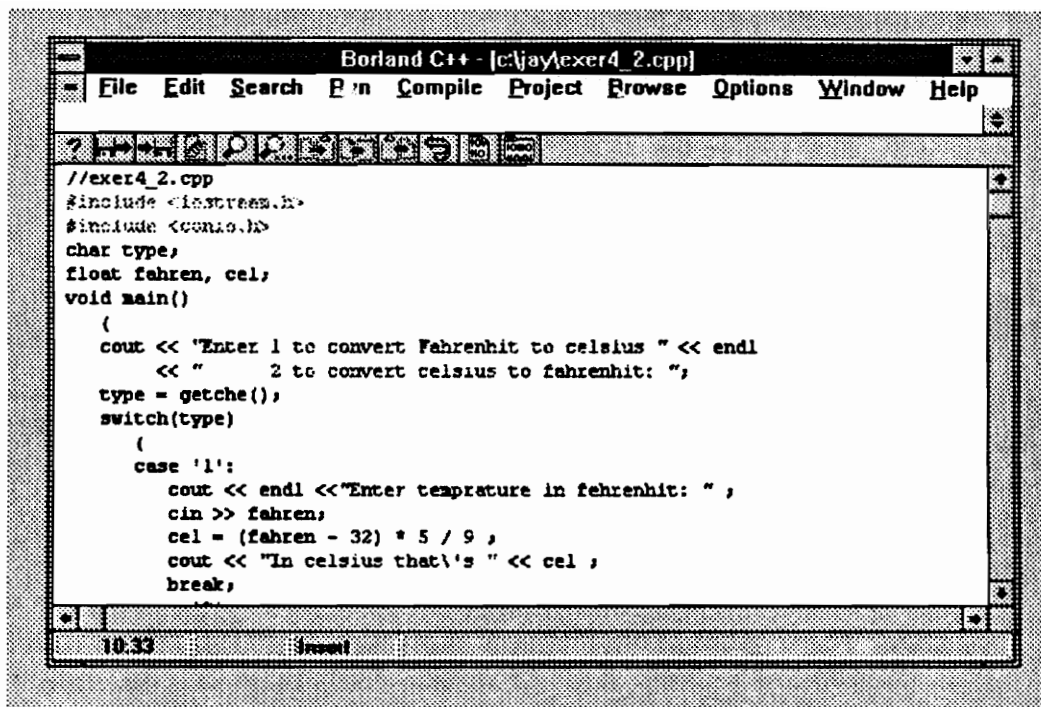


Figure 3.13 Borland C++ for Windows Integrated Environment

accelerator tables. The resource file can be created by using the Whitewater Resource Toolkit or the Resource Workshop.

The IDE also has menu choices for compiling and linking the application. All files in the project are compiled and linked to make an executable file. The executable file can then be executed to run the application independently in Windows environment. The Browser menu invokes a dialog box which displays the hierarchy of classes and methods. The methods can be browsed for debugging.

Another important utility in the IDE is the debugger. The debugger allows the execution of the program in steps. It also shows values of the variables during execution. This can be of immense help while debugging a program. By setting 'Watch' points, steps can be specified in the program execution. The debugger then executes the program in steps and the variables values can be checked at each step. The values can be modified and the program can be checked for the modified values.

Thus the Borland C++ IDE provides all the necessary tools for creating an executable file of a project. This is far superior to the Actor environment where the creation of an stand-alone application involves rigorous process.

3.3.3 The Object Windows Library

The Object Windows Library (OWL), available separately for Borland C++ for Windows, is a class library for developing Windows applications. The OWL contains a large variety of classes that correspond to interface objects in Windows. Some of the classes in OWL include classes for Windows, dialog boxes, buttons, scroll bars, combo boxes, bitmaps. Many of these classes respond to the Windows messages. These classes can be used directly or new classes can be derived from the these classes to create Windows applications. This saves a lot of efforts on the programmer's part. A derived class, defined in the application can be modified to respond to other Windows messages.

This gives the programmer an opportunity to improve the behavior of the Windows elements to suit the requirements of a specific application.

In the Object Windows Library, all window elements are considered as objects. The Object Windows Library provides encapsulation of window's information. This encapsulation makes it possible for each window object to encapsulate its data regarding the behavior and the attributes. The Object Windows Library also provides abstraction of most of the Windows API functions. This allows the programmer to use the object windows functions, instead of the Windows API functions. However, it is also possible to make direct calls to Windows API functions if required. The parameters required for making Windows function calls are encapsulated in the objects as data. Another feature of Object Windows Library classes is the automatic message response architecture which allows the programmer to write the functions in response to a particular Windows message. When a window object receives a message, a defined member function is called automatically. Thus the Object Windows Library allows the programmer to develop Windows application using object oriented techniques without having to know the details of many of the Windows API functions.

In this chapter, both the Actor and C++ programming languages were discussed. The main features of Actor programming include the Actor environment, the classes, objects and methods were presented. As Actor is not an integrated development environment the creation of an executable file has been separately presented in the section of 'Creating a Stand-alone Application'. Windows resources and WhiteWater Resource Toolkit were also discussed. The main features of C++ programming were presented taking into consideration classes, objects and methods. The Borland C++ integrated environment was presented followed by an overview of Borland C++ Object Windows Library. In the next two chapters the applications developed using Actor and C++ will be discussed.

Chapter 4

Actor Applications

4.1 Introduction

This chapter describes the applications developed using Actor. The programs developed using Actor were: a) Flexural Capacity of Tee Beam, b) Design of Simply Supported Reinforced Concrete Beams, and c) Design of Continuous Reinforced Concrete Beams.

The first two application programs, Flexural Capacity of Tee Beams and Design of Simply Supported Reinforced Concrete Beams, are relatively simple applications. The primary purpose of writing these applications was to develop an understanding of the use of object oriented programming concepts in structural engineering. It is interesting to note that it took approximately the same amount of time to develop the Tee Beam application as it did the Continuous Beam program. This is directly attributable to the fact that by the time the continuous beam program was developed, the author had developed considerable experience and insight in the application of object oriented technology to structural engineering.

In the sections that follow a detailed description of the three programs is presented. An overview of each application is presented first. This is followed by a description of the main classes and functions. Then the class structure of each application is presented. For the continuous beam application, the flow chart for each class is also presented.

4.2 Flexural Capacity of Tee beam

4.2.1 Overview of the Tee Beam Application

This program computes the flexural capacity of a tee beam in accordance with the ACI 318-89 specifications [33]. The input to the program consists of section properties of the beam, the area of reinforcing steel, and material properties such as, concrete compressive strength and yield strength of the reinforcing steel. The program computes the flexural capacity of the section and displays the results and input data on the screen.

The main menu of the application window consists of three menu items as shown in Figure 4.1.

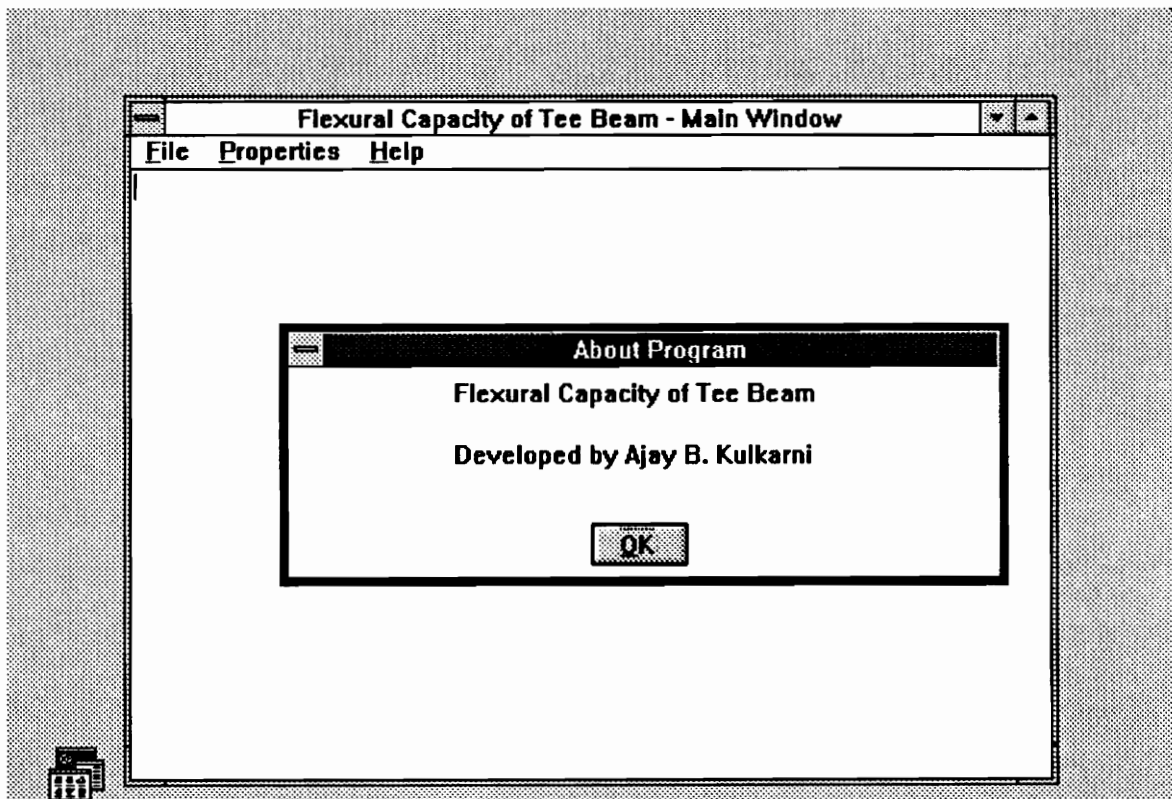
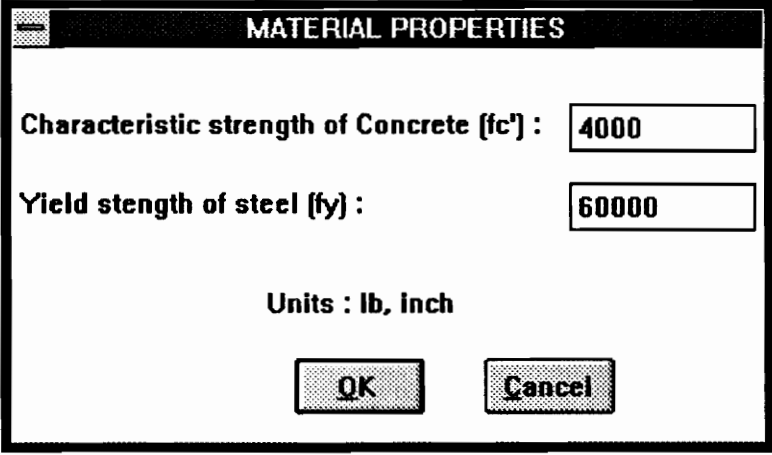


Fig. 4.1 Main Menu: Flexural Capacity of Tee Beam

The menu items are: 'File', 'Properties' and 'Help'. The 'File' menu has three choices: 'New', 'Print', 'Exit'. The 'New' sub menu if selected, creates a new data file. All data are initialized to zero and the main window is updated by calling the 'write' method. The 'Print' sub menu can be selected to print the current data. The 'Exit' menu terminates the application.

The data for the program are entered by selecting the 'Properties' menu. The 'Properties' menu has two sub menus: 'Material' and 'Section'. The selection of the sub menu 'Material' opens the material properties dialog box as shown in Figure 4.2.



The image shows a dialog box titled "MATERIAL PROPERTIES". It contains two input fields: "Characteristic strength of Concrete (fc') :" with the value "4000" and "Yield strength of steel (fy) :" with the value "60000". Below these fields, it says "Units : lb, Inch". At the bottom, there are two buttons: "OK" and "Cancel".

Fig. 4.2 Flexural Capacity of Tee Beam : Material Properties Dialog

The material properties dialog box is an object of the class 'AjTee'. The dialog box has two edit boxes one for the concrete strength and the other for the yield stress of steel (Fig. 4.2). The material properties data box is closed when either the 'OK' or the 'Cancel' button is pressed. When the user enters the data and presses the 'OK' button, the method 'okPressed' of the class 'AjTee' is called. This method in turn calls the method 'matIn'. The 'matIn' method checks if the values entered by the user are positive real numbers. If the values entered are incorrect then a message box pops up and the dialog box is not closed. Fig. 4.3 shows a typical message for invalid data.

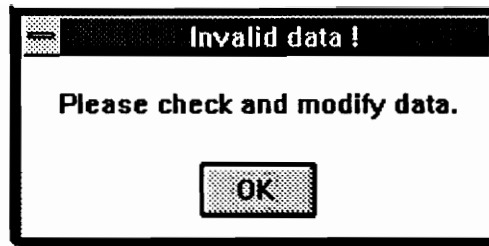


Fig. 4.3 Flexural Capacity of Tee Beam: Invalid data message box

If the material data input is correct then the values entered are assigned to the class variables \$fc and \$fy of the main window. The β_1 value as defined in Section 10.2.7.1 of the ACI 318-89 [33], is computed by the method 'betaCal' of the class 'AjTee'. If geometric data is also available, then the 'matIn' method calls the 'balBlock' method. The 'balBlock' method computes the depth of the neutral axis for the given section for the balanced condition. Then the 'balBlock' method calls the method 'balStrength' to check the section for ductility. The ductility check is carried out by the method 'balStrength' along with the method 'ductCheck'. The 'ductCheck' method calls the 'mmtCap' method to compute the flexural strength of the given section. The main window is updated to show the computed moment capacity. Thus, each time any data, either material properties or geometric properties, are modified, the main window updates the screen and displays the new moment capacity (Fig. 4.4).

Section properties of the section can be entered by selecting the 'Geometry' sub menu of the 'Properties' menu. A dialog box, as shown in Fig. 4.5 is displayed. The dialog box object called 'sectWindow', is an object of the class 'AjTee'. The data for the width of the flange, width of the web, effective depth, depth of flange and area of tension steel is entered in units of lb and inches. When the 'OK' button is selected the 'okPressed' method of the 'AjTee' class calls the method 'sectIn'. The 'sectIn' method checks the data. The method then updates the main window. If the material properties are also available then the moment capacity of the section is computed. If the user presses the 'Cancel'

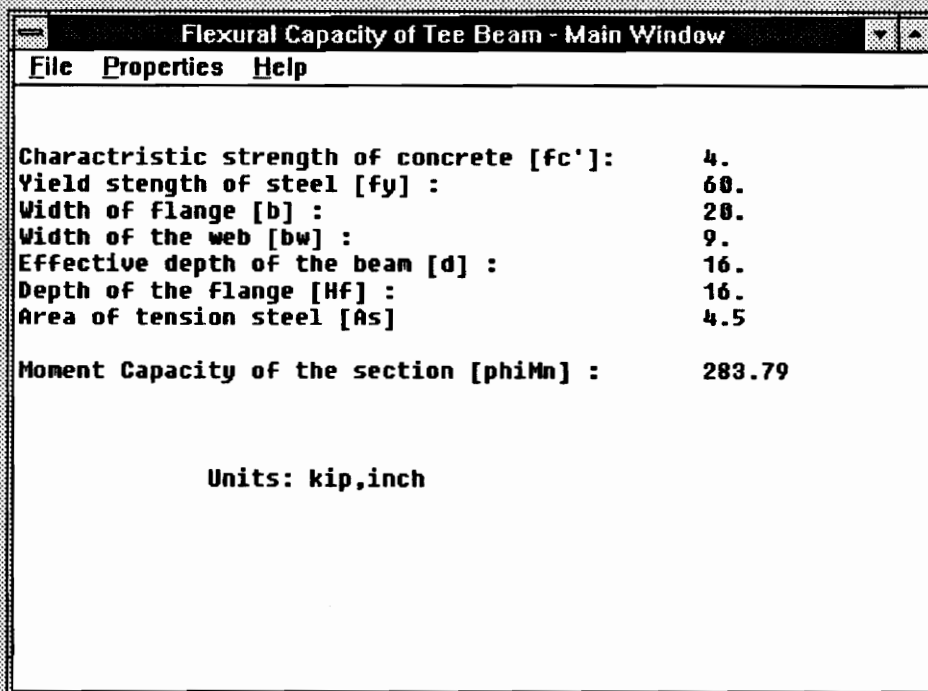


Fig. 4.4 Flexural Capacity of Tee Beam: Output Screen

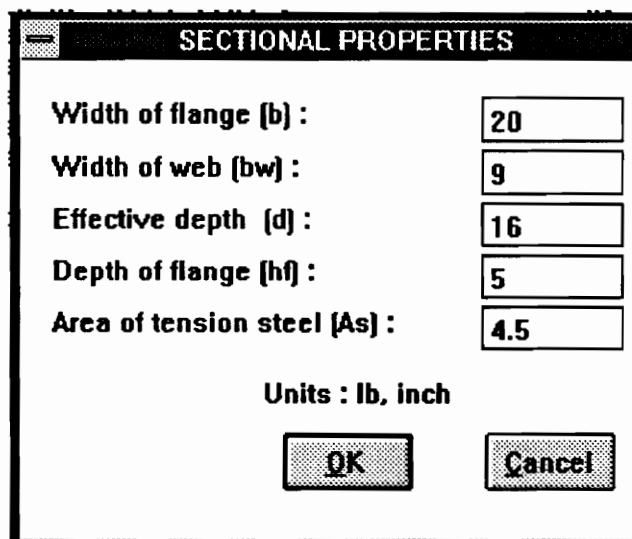


Fig. 4.5 Flexural Capacity of Tee Beam: Section properties dialog box

button at any time during data entry, the dialog box is closed without updating the data.

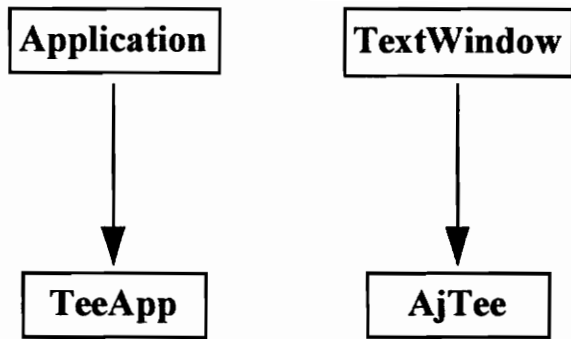
4.2.2 Class Structure

Only two classes are created for the Tee Beam Application program: class 'AjTee' and class 'TeeApp' (Fig. 4.6). The class 'TeeApp' is derived from the Actor class 'Application'. The method 'initMainWindow' creates the main window object of the class 'AjTee'. The listing of the 'initMainWindow' method is shown below.

```
/* Flexural capacity of tee beam - application initialization */
Def initMainWindow(self)
{
  mainWindow := loadTopFrom(AjTee,"maintee");
  setIcon(mainWindow,IDI_APPLICATION);
  reorientMain(mainWindow);
}
```

The 'initMainWindow' method creates the main window object as defined in the "maintee.wdl" attribute definition file. The attribute definition file attaches the menu attributes to the main window. The menu is created using the Actor Resource Toolkit. The main window is an object of the class 'AjTee'. The last statement of the 'initMainWindow' method calls the 'reorientMain' method of the object 'mainWindow'. This 'reorientMain' method is defined in the class 'AjTee' and defines the location of the main window.

The class 'AjTee' was a sub class of the library class 'TextWindow'. Fig. 4.6 shows the data variables declared for this class. The 'TextWindow' class was used as the base class since it provided methods to display the data on the main window. The main window and the dialog box objects were developed from this class. This ensured the availability of the data members of the application to all the dialog boxes and the main window. Since this is a small application only three dialog boxes are required: the About dialog box, the Material Properties dialog box, and the Section Properties dialog box. The "About" dialog box, displays information about the application. It is activated by selecting



Variables :

mainWindow

b, bw, d, hf, As, fc, fy
Nc, phiMn, Beta,
maxWidth, maxHeight

Fig. 4.6 Class Structure for Tee Beam Application

the 'About' menu item from the 'Help' menu. The material properties dialog box and the section properties dialog box obtain user input. The main window object computes the capacity of the section each time there is a change in material or section properties. The class 'AjTee' has the following important methods:

```
balBlock(self)          /* Computes the stress block of the
                        balanced section */
balStrength(self)       /* Computes the capacity of the balanced
                        section */
ductCheck(self)         /* Performs the ductility check */
betaCal(self)           /* Computes  $\beta$  for given material */
mmtCap(self)            /* Computes the moment capacity of the
                        given section */
write(self)             /* Updates the data in main window */
```

The 'mmtCap' method also calls the method 'write' that modifies the main window which displays the input data and the moment capacity of the section.

This program is a very useful tool for computing the flexural capacity of tee sections. The ability to modify the section capacity each time the section or material properties are changed is a valuable asset. This program was fairly simple to develop using object oriented programming techniques. However, it proved to be a very good starting application for understanding object oriented programming concepts. The application was useful for learning the techniques required for creating objects for representing dialog boxes. This knowledge was useful in the development of the other applications.

4.3 Design of Simply Supported Beam

4.3.1 Overview of the Simply Supported Beam Application

This program designs a simply supported reinforced concrete beam subjected to as many as ten load combinations and ten load cases. The program can handle the following loads :

1. Uniformly distributed loads over entire span;
2. Partially distributed loads;
3. Concentrated loads;
4. Linearly varying loads;
5. Moments applied at the supports.

The program can handle up to five loads of each type except for concentrated loads. A maximum of ten concentrated loads are allowed. The data for the program can be entered by selecting the appropriate menu items from the main window of the application. The selection of each sub menu from the main menu causes the creation of the corresponding dialog box for data entry. The 'File' menu of the main window of the application (Fig. 4.7) consists of following sub menu items :

<u>F</u> ile -	<u>N</u> ew -	Selected to start a new data file.
	<u>O</u> pen-	Open an existing data file.
	<u>S</u> ave-	Save the current data file.
	Save <u>A</u> s-	Save the current data in a file.
	<u>P</u> rint-	Prints the file on a connected printer.
	<u>E</u> xit-	Exits the application.

The sub menu 'Geometry' of the 'Properties' menu opens a dialog box as shown in Fig. 4.8. The span can be entered in feet, in the edit control of the span dialog box. The beam can be designed either for external exposure or for internal exposure as per ACI 318-89 specifications [33]. The exposure condition can be selected from the list box in the dialog box.

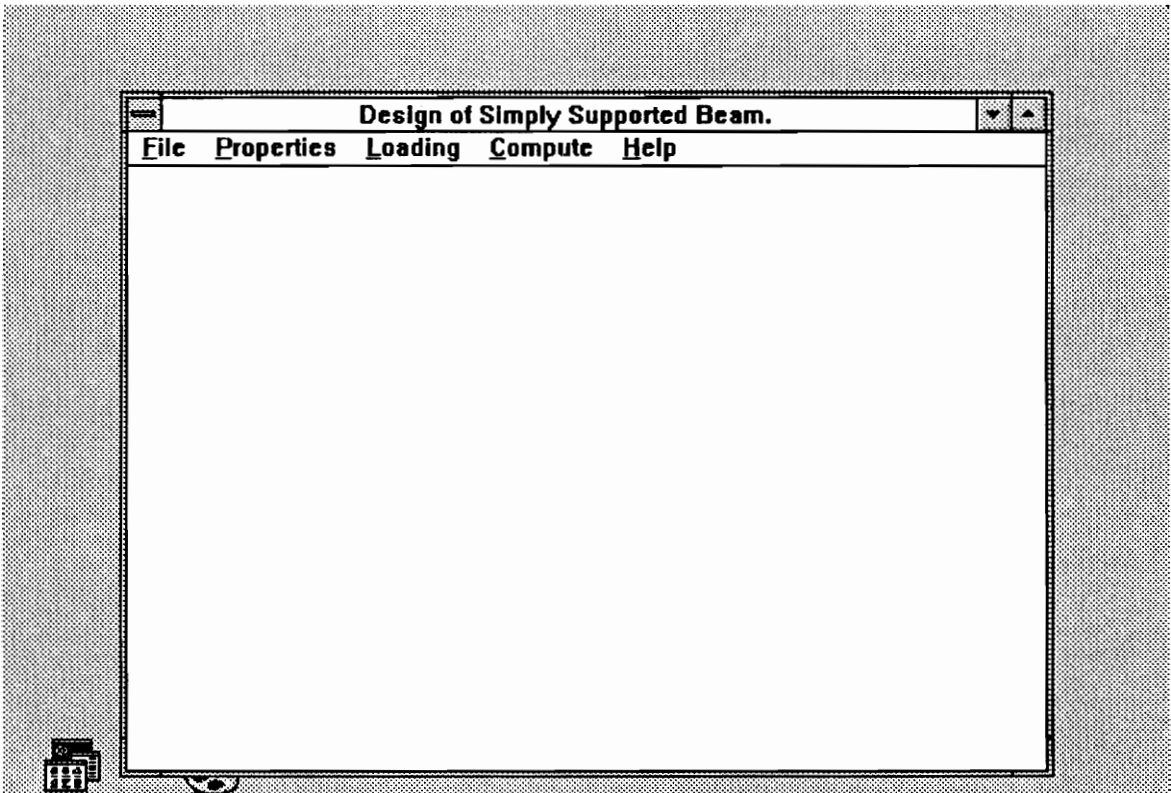


Fig. 4.7 Design of Simple Beam: Main Window

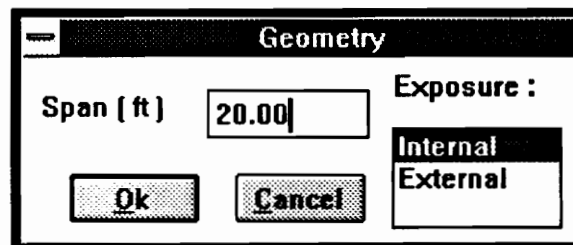
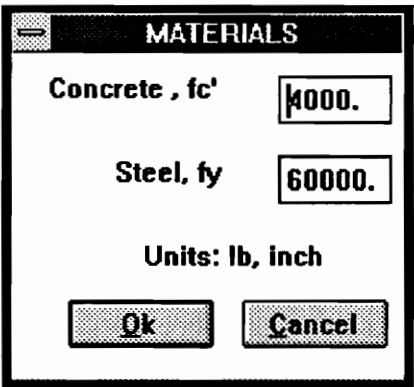


Fig. 4.8 Design of Simple Beam: Geometric Properties

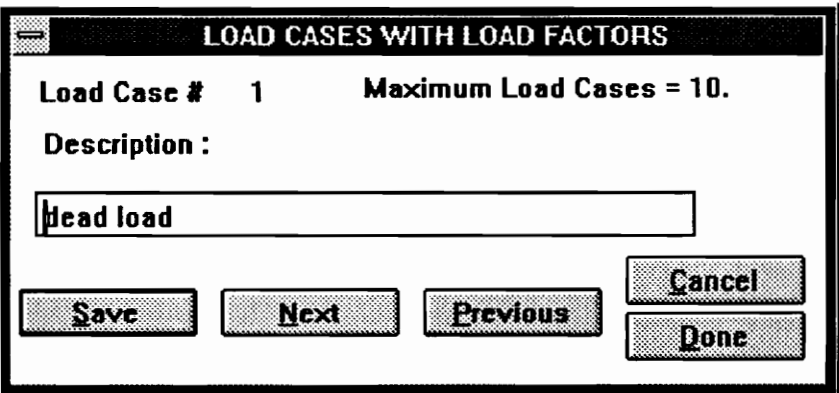
The sub menu 'Material' of the 'Properties' menu opens a dialog box as shown in Fig. 4.9. The data consist of the compressive strength of concrete (f_c') and the yield strength of the reinforcing steel (f_y). The values are entered in psi.



A dialog box titled "MATERIALS" with a standard Windows-style title bar. It contains two input fields: "Concrete, f_c' " with the value "4000." and "Steel, f_y " with the value "60000.". Below these fields, it says "Units: lb, inch". At the bottom, there are two buttons: "Ok" and "Cancel".

Fig. 4.9 Design of Simple Beam: Material Properties

The load data for the beam are entered through the sub menus of the 'Loading' menu. The Load Case descriptions, such as Dead Load, Live Load, Wind Load, etc. are entered by selecting the 'Load Cases' sub menu. The dialog box for entering load cases descriptions is shown in Figure 4.10.



A dialog box titled "LOAD CASES WITH LOAD FACTORS" with a standard Windows-style title bar. It displays "Load Case # 1" and "Maximum Load Cases = 10.". Below this, it says "Description :". There is a text input field containing "Dead load". At the bottom, there are five buttons: "Save", "Next", "Previous", "Cancel", and "Done".

Fig. 4.10 Design of Simple Beam: Load Cases

Up to ten load cases can be handled by the program. The 'Next' and 'Previous' buttons in the dialog box can be used to move from one load case to another. The Load Cases dialog box specifies the names of the load cases used in the program (Fig. 4.10).

Various loads acting on the beam are entered by selecting the appropriate sub menu from the 'Load Type' menu. Figure 4.11 shows the dialog box for distributed load over the entire span.

Distributed Load
Positive Downward

Intensity (w) in kip/ft = 4.115

Select load case :

- dead load
- live load
- test

Buttons: Cancel, Done, Save, Next, Previous

Fig. 4.11 Design of Simple Beam: Distributed Load Dialog Box

The load intensity is entered in kip/ft in the edit box. The load case list box contains the list of all the load cases specified by the user and a load case can be selected from the list. Since the load intensity data requires information such as Load Cases and span, load data cannot be entered before entering span data and load case data. The user is required to enter data in the order given by the menus in the main window. Thus, before entering loads, beam geometry, material properties and load case descriptions have to be entered. The load data entry format is kept similar for all of the load types. Depending upon the type of load additional data may be required (see Figures 4.11 - 4.15). The 'Next' and 'Previous' buttons in the dialog boxes can be used to move from one

Partial Distributed Load
Positive Downward

Load Intensity, w , (kip/ft)

Distance, a , from left support (ft)

Spread, c , of the load (ft)

Select load case :

dead load	+
live load	
test	
	+

Fig. 4.12 Design of Simple Beam: Partially Distributed Load Dialog Box

Concentrated Load
Positive Downward

Concentrated Load, P , (kip)

Distance, a , from left support (ft)

Select load case :

dead load	+
live load	
test	
	+

Fig. 4.13 Design of Simple Beam: Concentrated Load Dialog Box

Moment at left support

Positive Clockwise

Moment, M, (kip-ft)

20

Select load case

dead load

live load

test

+

+

Cancel

Done

Save

Next

Previous

Fig. 4.14 Design of Simple Beam: Left Support Moment Dialog Box

Moment at right support

Positive Counter-Clockwise

Moment, M, (kip-ft)

-20.0

Select load case :

dead load

live load

test

+

+

Next

Cancel

Previous

Done

Save

Fig. 4.15 Design of Simple Beam: Right Support Moment Dialog Box

load to another (Fig. 4.11 - 4.15). The selection of the 'Cancel' button abandons the data entry and closes the dialog box.

The load combination dialog box shown in Figure 4.16 below allows the user to enter load factors for each load case in the load combination.

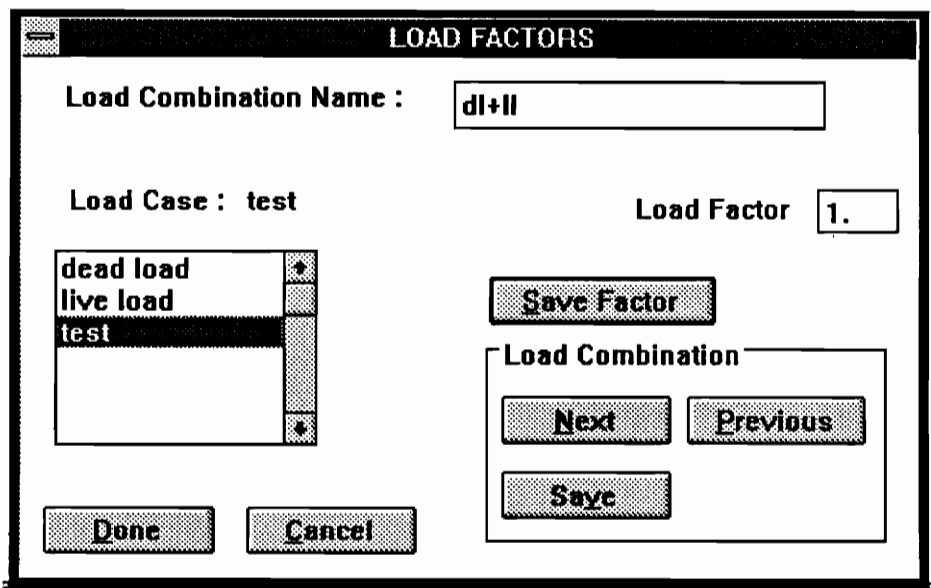


Fig. 4.16 Design of Simple Beam: Load Combination Dialog Box

Up to ten different load combinations can be entered. For example, as shown in Fig. 4.16, a load case 'test' can be given a load factor of 1.0 in the Load Combination dialog box. The 'Save Factor' button saves the load factor. The program can handle up to ten different load combinations. The program computes the worst load combination corresponding to the given load factors.

The design process can be started by selecting the 'Flexural Design' sub menu from the 'Compute' menu. The program computes the worst load combination effect and designs the beam section for the absolute maximum applied moment. Shear and moment computations are performed at seven sections along the span. The maximum moment for

each of these sections is computed. The beam dimensions are computed based on the absolute maximum moment. The program displays a list of acceptable sections based on these computations. The recommended sections are such that they all have a depth to width ratio in the range of 1.4 to 2.00. Figure 4.17 shows the section selection dialog box with the recommended sections.

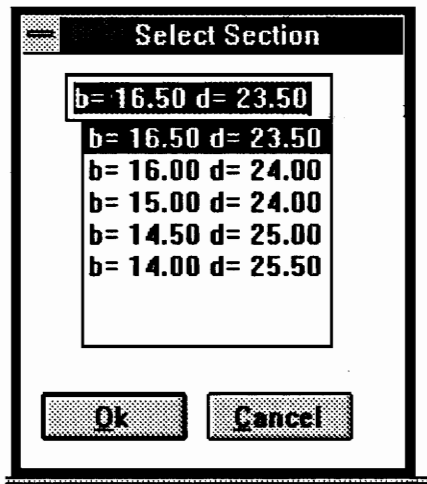


Fig. 4.17 Design of Simple Beam: Recommended Section

The flexural steel requirements for the selected section are computed and the recommended values of bar sizes and bar numbers are displayed. Fig. 4.18 is a typical dialog box displaying the recommended tension steel reinforcement.

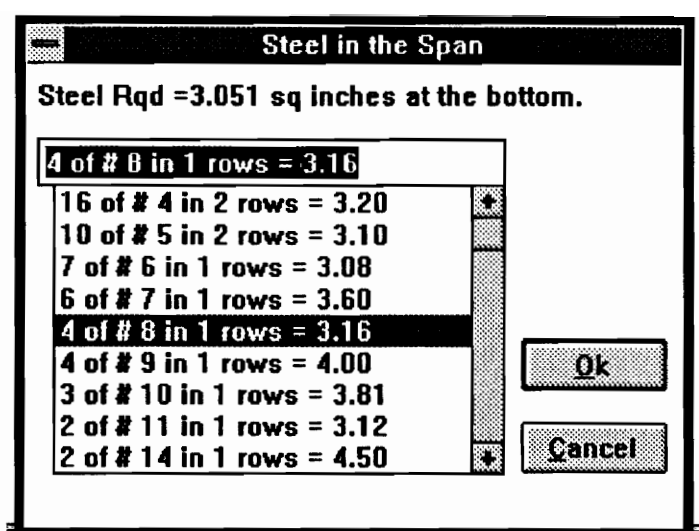


Fig. 4.18 Design of Simple Beam: Flexural Steel Dialog Box

The shear design begins with the user's choice of bar size as shown in Figure 4.19.

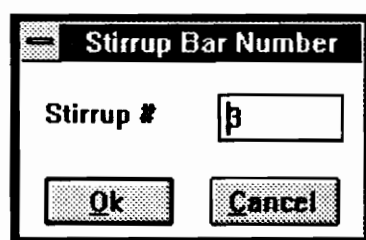


Fig. 4.19 Design of Simple Beam: Stirrup Bar Dialog Box

The Stirrup spacing is computed at all the seven sections and the results are displayed on the screen as shown in Fig. 4.20.

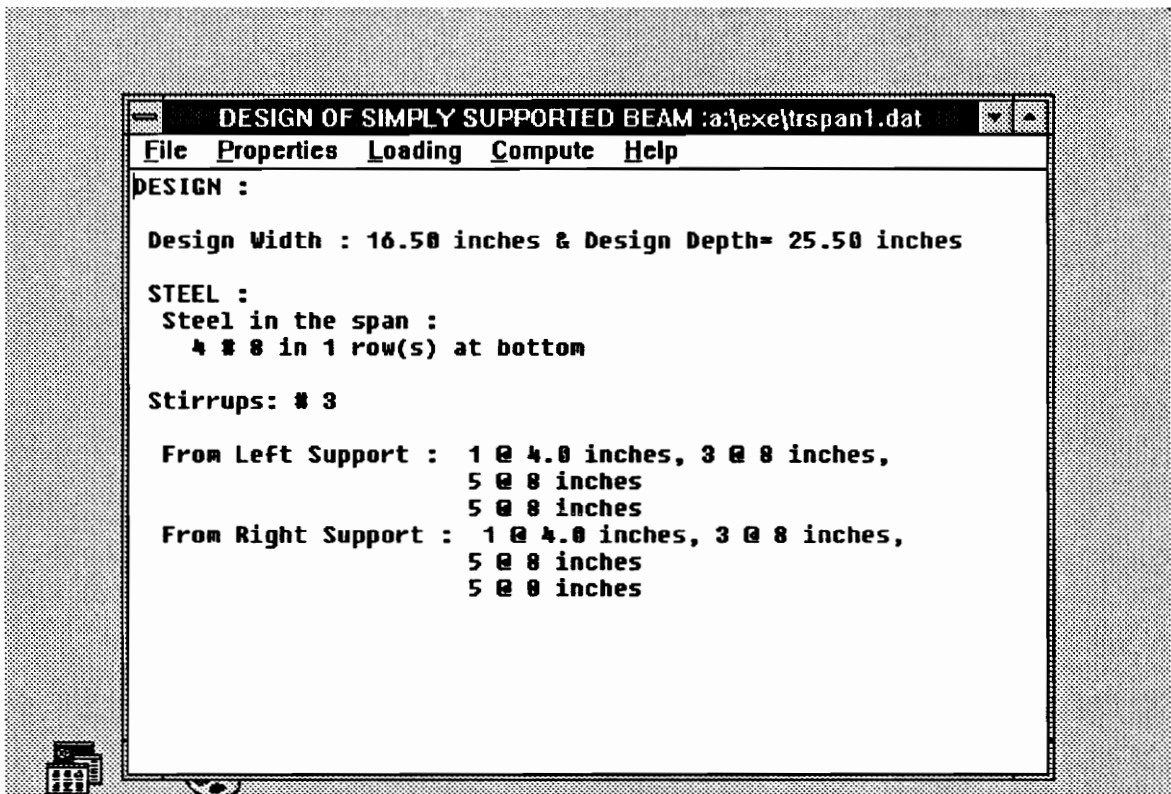


Fig. 4.20 Design of Simple Beam: Results

4.3.2 Class Structure

The class structure for the Simple Beam application is shown in Fig. 4.21. The class structure shows the application class 'SimpleApp', which is derived from the Actor 'Application' class. This class initiates the application by creating the main window of the application and then calls the 'reorientMainWindow' method which positions the main application window at the center of the screen.

The class 'Load', a sub class of the Actor library class 'TextWindow', serves as the class for the main window object. All the data members required for the design are declared as class variables within the 'Load' class. Fig. 4.21 also lists the data variables for each class. The main window object basically obtains input from the user through several dialog boxes. These dialog boxes are objects of the class 'Load' and therefore have the

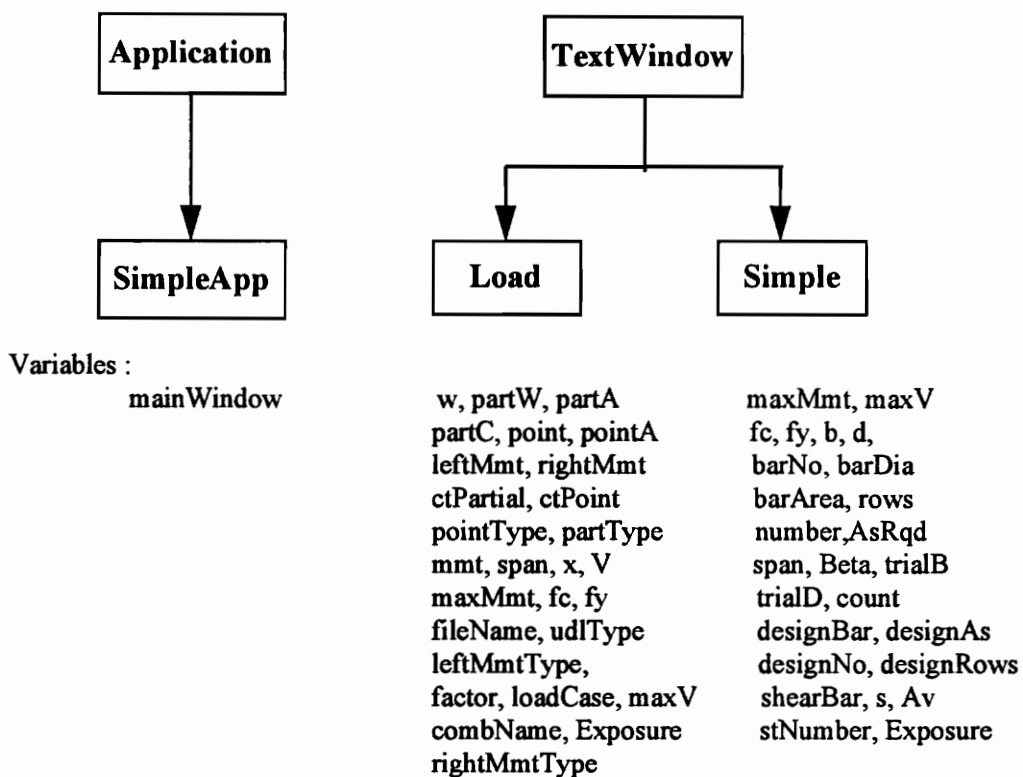


Fig. 4.21 Class Structure for Simple Beam Application

same data members.

The selection of the 'Flexural Design' menu creates an object 'simpObject' of the class 'Simple'. Data is transferred to the 'simpObject' object and then the methods for performing the design computations in 'simpObject' are called. Fig. 4.22 shows the flowchart for the object 'simpObject' and the design procedure.

The Simple Beam application described in this section had several limitations. First, the design process divides the beam span in only seven sections for moment and shear computations. It is quite possible that the computed maximum moment or shear section may not be accurate since the maximum moment or shear may lie between the points at which the shear and moment values are computed. Another limitation is that the section dimensions have to be selected from the list of sections provided by the program. There is no option for selecting a section other than the ones listed. The same is true of reinforcement. The reinforcement has to be selected from the values given in the table. These limitations can be overcome by modifying the program. The continuous beam application does not have these limitations and provides considerable flexibility in the selection of section dimensions and reinforcement.

Despite these limitations, this application is valuable in illustrating the basic concepts of the application of object oriented programming to concrete design. It also provided the class 'Simple' which was used with some modifications, in the development of the program for the analysis and design of reinforced concrete continuous beams.

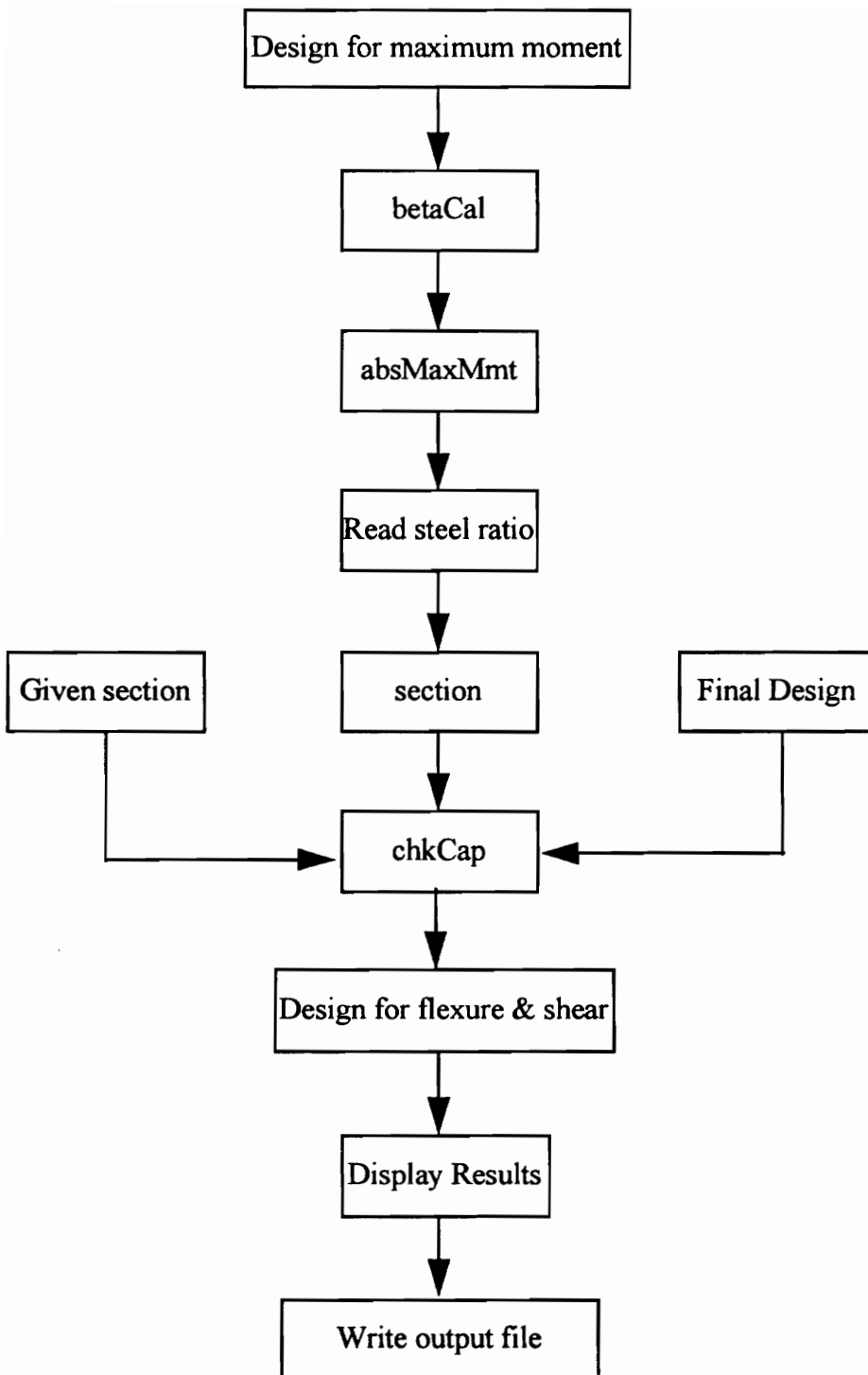


Fig. 4.22 Flow-Chart of object 'simpObject'

4.4 Analysis and Design of Reinforced Concrete Continuous Beam

4.4.1 Overview of the Continuous Beam Design Application

The third application developed was a program to design reinforced concrete continuous beams. This program was developed with the intention of applying object oriented programming techniques and testing the advantages of this techniques in a real world situation. The program performs the analysis and design of a continuous beam having a maximum of 12 spans, subjected to a maximum of ten load cases, and ten load combinations. The program can handle the following types of loads :

1. Uniformly distributed load over the entire span;
2. Partially distributed load;
3. Concentrated load;
4. Linearly varying load.

The program can handle up to ten concentrated loads and five of each of the other types of loads, for each span. The program can also handle settlement of the supports.

Data for the continuous beam can be given by selecting the sub menus in the main window. Fig. 4.23 shows the main window of the application along with the About dialog box. The About dialog box tells the user about the program and can be opened by selecting the 'About' sub menu in the 'Help' menu.

The data for the program is divided into two distinct types, properties and loading. The 'Properties' menu of the main window handles the data entry of geometric data and material properties. The Geometric Data dialog box is shown in Fig. 4.24. This dialog box is similar to the corresponding dialog box in the simply supported beam design program (Fig. 4.8) except that it also has 'Next' and 'Previous' buttons for entering data for consecutive spans. The moment of inertia is used in the analysis of the continuous beam. The 'Done' button closes the dialog box and transfers the data to the main window object.

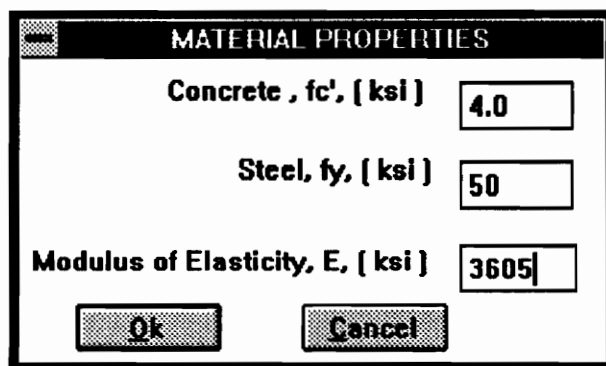
Design of Reinforced Concrete Continuous Beam						
File Properties Loading Analysis Design View Help						
Span # 1: B =10. in D =17. in						
Stirrups: #3 Max. Perm. As =3.85 sqin Min. Perm. As =0.56 sqin						
x(ft)	Steel (sqin.)		Stirrups	x(ft)	Steel (sqin.)	
	Top	Bottom	space(in)		Top	Bottom
						space(in)
0.00	None		7	0.66	0.22	7
1.33	0.43		7	2.00	0.63	7
2.66	0.82		7	3.33	1.00	7
4.00	1.16		7	4.66	1.31	7
5.33	1.45		7	6.00	1.56	7
6.66	1.67		7	7.33	1.75	None
8.00	1.81		None	8.66	1.86	None
9.33	1.89		None	10.0	1.90	None
10.6	1.89		None	11.3	1.86	None
12.0	1.81		None	12.6	1.75	None
13.3	1.67		7	14.0	1.56	7
14.6	1.45		7	15.3	1.31	7
16.0	1.16		7	16.6	1.00	7
17.3	0.82		7	18.0	0.63	7
18.6	0.43		7	19.3	0.22	7
20.0	None		7			

Fig. 4.23 Design of Continuous Beam: Main Window

SPAN DATA	
Maximum Number of Spans = 12	
Span #1 (ft) :	<input type="text" value="20."/>
Moment of Inertia (in ^ 4) :	<input type="text" value="4000."/>
<input type="button" value="Next"/>	<input type="button" value="Previous"/>
<input type="button" value="Done"/>	

Fig. 4.24 Design of Continuous Beam: Geometric Properties

The Material Properties dialog box as shown below can be opened by selecting the 'Material' sub menu of the 'Properties' menu.

A screenshot of a software dialog box titled "MATERIAL PROPERTIES". The dialog box has a dark title bar with a small icon on the left. Inside, there are three input fields with labels to their left: "Concrete , fc', (ksi)" with a value of "4.0", "Steel, fy, (ksi)" with a value of "50", and "Modulus of Elasticity, E, (ksi)" with a value of "3605". At the bottom of the dialog box, there are two buttons: "Ok" and "Cancel".

MATERIAL PROPERTIES	
Concrete , fc', (ksi)	4.0
Steel, fy, (ksi)	50
Modulus of Elasticity, E, (ksi)	3605
<div>Ok Cancel</div>	

Fig. 4.25 Design of Continuous Beam: Material Properties Dialog Box

The dialog box is used to input the compressive strength of the concrete and the yield stress for steel. The modulus of elasticity in this dialog is used in the analysis of the beam. The units are inch and kip.

The Load Cases dialog box, which is similar to the corresponding dialog box for the simple beam design application shown in Fig. 4.10, is used to specify load cases. A maximum of ten load cases can be specified for a continuous beam. A load case name can be any combination of alphabets and numbers. These load cases are used to specify the load factors in the Load Combination dialog box (Fig. 4.26).

The Load Combination dialog box is shown in Figure 4.26. The 'Add Case' and 'Delete Case' buttons are used to specify load factors. The load factor for a given load case can be entered in the 'Load Factor' edit box. This load factor corresponds to the load case selected in the list box. By selecting a load case from the list box and entering the load factor, the user can specify load factors for all load cases in a particular load combination. The user can move from one load combination to another by using the

'Next' and 'Previous' buttons (Fig. 4.26). The 'Done' button closes the dialog box and transfers the data from the dialog box to the main window.

LOAD FACTORS

Load Combination :

comb 1

Load Case :
live load

Load Factor :
1.0

live load

Load Combination

Next

Previous

Done

Add Load Case

Delete Load Case

Fig. 4.26 Design of Continuous Beam: Load Combinations

Loads can be entered by selecting from the 'Load Type' menu. Figure 4.27 shows a typical dialog box for entering distributed load. The intensity of the distributed load over the entire span can be entered in the edit box. The load case for the load and the span can be selected from the list boxes provided for this purpose. This facilitates entering loads for any span. The 'Next' and 'Previous' button permit forward and backward movement in the load data table for the selected span.

Fig. 4.27 Design of Continuous Beam : Distributed Load

There are three steps in the design of the beam. The first step is the analysis of the beam for the given loads and span. The analysis of the continuous beam is performed using the matrix method and Holzer's approach [27]. After the analysis is completed the design is started by choosing the 'Preliminary Design' menu. During the preliminary design several trial sections are determined. The preliminary design has two menu choices :

1. From the 'Given Section' menu, it is possible to specify the beam section. The dialog box for entering section dimensions is shown in Figure 4.28.

Fig. 4.28 Design of Continuous Beam: Given Section Properties

This option makes it possible to limit the size of the section, for example, in situations where it may be necessary to limit the depth of the beam for architectural reasons. This option can also be used when the cross sectional dimensions are known in advance, from previous experience, or from an earlier design done using the 'Design for Maximum Moment' option.

2. When the 'Design for Maximum Moment' menu option is selected, the program determines the required sections for a given steel ratio. The steel ratio can be entered in the Steel Ratio dialog box shown in Fig. 4.30. The dialog box shows the minimum and maximum possible steel ratios for given material properties according to the ACI 318-89 specifications [33]. The dialog box also shows the recommended value of the steel ratio which is an average of the minimum and maximum steel ratios.

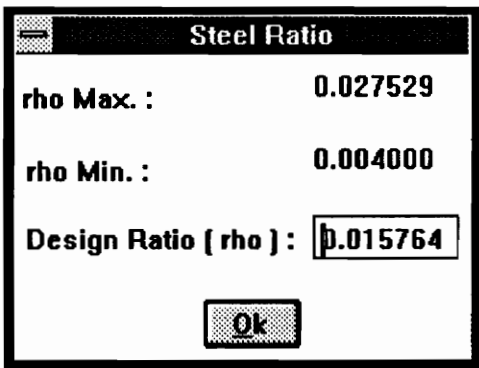


Fig. 4.29 Design of Continuous Beam: Design Steel Ratio Dialog Box

After the steel ratio is entered, the program computes the required section dimensions for the worst load combination. For the moment and shear computations, each span is divided into thirty equal sections and the maximum moment in each span is

computed. The moment computations are repeated for all load combinations. The maximum moment at each section (from all load combinations) is computed. The maximum moments for each span are compared to determine the maximum moment along the entire continuous beam. The required section is then determined for this absolute maximum moment using the ACI 318-89 specifications. A range of sections is determined such that the ratio of depth to width for these sections lies between 1.40 and 2.0. The dialog box as shown in Fig. 4.30 displays the possible beam sections. It is possible to choose a section from the list of recommended sections or to enter the dimensions of any other section.

SECTIONAL PROPERTIES

Recommended
Section :

9.50, 14.5

10.0, 14.0

9.50, 14.5

9.00, 15.5

8.50, 15.5

All Dimensions in
inches

Width

9.5

Depth

14.5

OK

Cancel

Fig. 4.30 Design of Continuous Beam: Design Section

The steel requirements and stirrup spacing requirements are then computed for this section at thirty-one equally spaced points along each span. The results are displayed as shown in the Fig. 4.31. The menu choices of 'Next Span' and 'Previous Span' from the 'View' menu can be used to view design results for each span. The output consists of the sectional dimensions, required area of flexural steel, and bar sizes and stirrup spacing at thirty one sections along the span (inclusive of supports). Once the dimensions of the

beam have been finalized, the 'Final Design' menu item can be chosen. When this option is selected, the program performs the flexural and shear design for the selected beam section. Again each span is divided and the design is performed at 31 sections. An output file is created and the data and the results are written to this output file. It should be noted here, that the sections may be doubly reinforced. The program computes appropriate area of tension and compression steel depending upon the type of section.

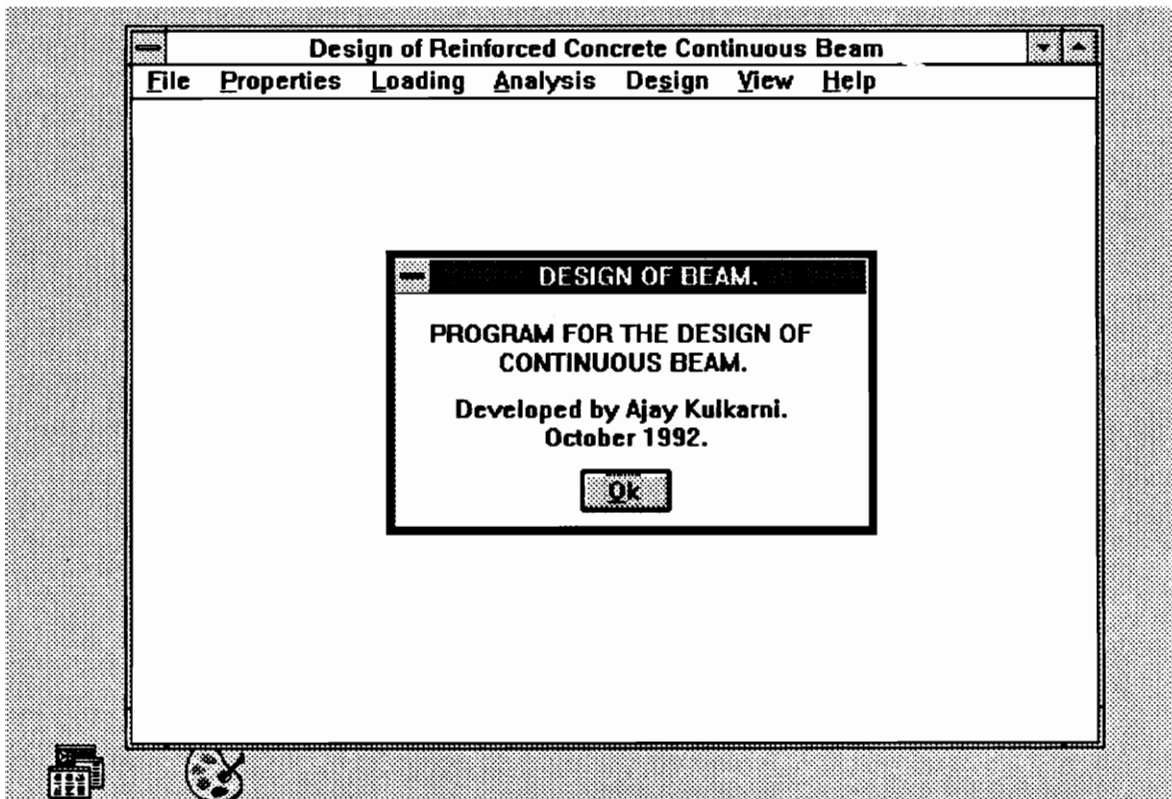
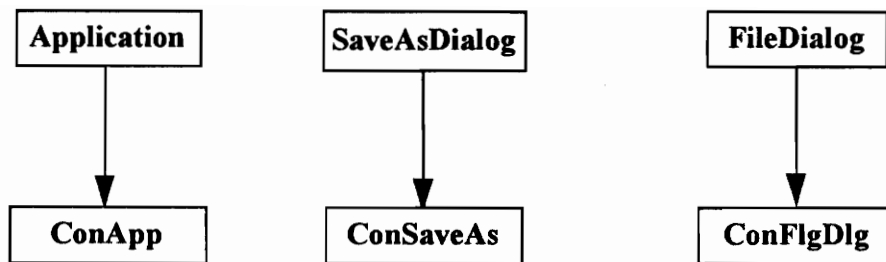


Fig. 4.31 Design of Continuous Beam: Results

4.4.2 Class Structure

The class structure for the program is shown in the Fig. 4.32. The topmost class in each branch is a class supplied by Actor. The other classes have been derived from Actor



Variables :
mainWindow

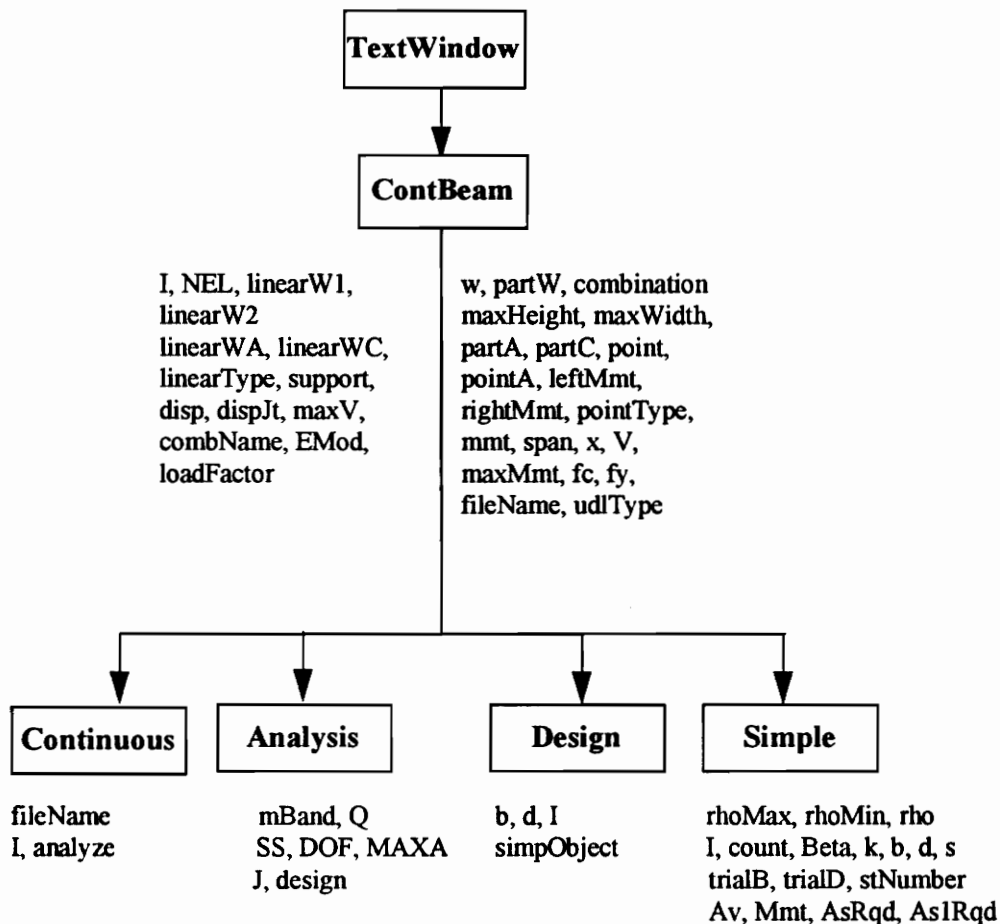


Fig. 4.32 Class Structure for Continuous Beam Application

classes for this application. The continuous beam application has eight classes that have been derived from the class library supplied with Actor.

The class 'ConApp', derived from the class 'Application' opens the main window of the application. The main window is an object of the class 'Continuous'. The 'ConApp' class has only one method, 'initMainWindow', which opens the main window. The 'initMainWindow' method calls the 'reorientMainWindow' method of the main window, which positions the main window in the center of the screen.

The class 'ConFlgDlg' derived from 'FileDialog' and the class 'ConSaveAs' derived from 'SaveAsDialog' provide the response methods for the file input/output menu selection. The required dialog boxes are provided in Actor's library of dialog box attribute files.

The class 'ContBeam', derived from the class 'TextWindow', is the main class of the continuous beam application. All the data members, common to the classes 'Continuous', 'Design', 'Analysis' and 'Simple' are declared as instance variables in the 'ContBeam' class (Fig. 4.32). This means that each object of these classes will have a separate set of these variables, thus preserving data security.

The ContBeam class has only one method, 'warn1'. This method is common to all the sub classes. This method opens a dialog box, displaying a warning message, if an attempt is made to quit any data entry dialog box without saving the data.

When the application is started, the main window of the application is opened as an object of the class 'Continuous'. This class has the data variables inherited from the class 'ContBeam'. In addition, the class 'Continuous' declares several instance variables such as, 'fileName', 'i', 'analyze'. The variable 'fileName' contains the name of the data file. The variable 'i' is a counter used to count the number of times data are entered. The variable 'analyze' is an object of class 'Analysis', and is created when the analysis is started. There is only one main window open at any given time. The 'mainWindow' object

provides the necessary user interface for entering data via menus and for displaying results.

The dialog boxes are created using the resources defined in a resource file. The attributes of the dialog boxes are stored in attribute files. Fig. 4.33 contains the listing of the attribute file "Geometry.wdl" which is used for the Geometrical Property dialog box.

```
Window Geometry
class Continuous
resource : "Geometry"
focus : span

Child Span
class : Edit
resource : 103

Child inertia
class: Edit
resource : 109

Child spanName
class : Static
resource : 102
data : "Span #1 (in ft)"

Child OKButton
class : Button
resource : 110

Child PrevButton
class : Button
resource : 105

Child Done
class : Button
resource : 107
```

Fig. 4.33 "Geometry.wdl" file

As Fig. 4.33 shows, the dialog box 'Geometry' is built as an object of class 'Continuous' by using the resource definition contained in the file 'Geometry.wdl'. The resource file contains information about the attributes of the dialog box, its size, location and child controls.

Each child control is linked to the attribute definition by its resource id. Thus, the child edit control 'span' in the attribute file is linked to the edit control with id 103 in the resource file. This child is then referenced in the methods of the class 'Continuous' of the dialog box 'Geometry', as child 'span'. The method shown in Fig. 4.34 , which is an action method of the class Continuous, obtains input for the next span when the 'Next' button is pressed.

```

/* Method to obtain span and inertia for next span */
Def nextSpan(self)
{
if asReal(getText(self[#span])) and asReal(getText(self[#inertia]))
then
    if asReal(getText(self[#span])) > 0 and
        asReal(getText(self[#inertia])) > 0
    then
        span[i] := asReal(getText(self[#span]));
        I[i] := asReal(getText(self[#inertia]));
        if i < 12
        then
            i := i + 1;
            setData(self[#spanName], "Span #"+asString(i+1)+" (ft) :");
            setData(self[#span], asString(span[i]));
            setData(self[#inertia], asString(I[i]));
        else
            errorBox("STOP", "Maximum number of Spans = 12");
        endif;
    else
        errorBox("STOP", "Negative or Zero data !");
    endif;
else
    errorBox("STOP", "Invalid data !");
endif;
setFocus(self[#span]);
}

```

Fig. 4.34 Method 'nextSpan'

In Fig. 4.34, the statement

```
span[i] := asReal(getText(self[#span]));
```

reads the text from the child edit control 'span' and converts it into a real number and assigns the result to span[i]. Since the dialog box is created as an object of the class

'Continuous' it can respond to the methods declared in the class 'Continuous' and also has data members inherited from the class 'ContBeam'.

Using a similar approach to that used for entering span data, the main window of the class 'Continuous' allows the user to enter all the necessary data through dialog boxes. It should be noted that each dialog box is a separate object of the class 'Continuous'. Thus each has its own set of the data. This preserves the individual identity of the dialog boxes and also ensures isolation of the main window from the dialog boxes data hence reducing the risk of accidental data corruption.

The main window creates an object 'analyze' of the class 'Analysis' when the 'Analysis' menu is selected. The class 'Analysis' is derived from the class 'ContBeam' and has the basic data inherited from it. In addition to this data, it has data members required for the analysis. Fig. 4.32 shows the variables declared for the class 'Analysis'.

When the 'Analyze' object is created, the 'send1' and 'send2' methods are called. These methods, defined for the class 'Analysis', transfer the data entered by the user and stored in the mainWindow object, to the object 'analyze'. The analyze object has methods for assembling the global stiffness matrix, assembling the global load vector and solving the resulting equations $kq = Q$ [27]. Fig. 4.35 shows the key methods of the class 'Analysis'.

The 'Analyze' object computes the support moments at the left and right supports in each span for all load combinations. It also creates a 'design' object of the class 'Design', which performs the design of the beam. The 'design' class, a sub class of the class 'ContBeam', has 'simpObject' and 'i' as its data members in addition to the data inherited from 'ContBeam'. The variable 'i' is a counter while 'simpObject' is an object of the class 'Simple'.

The 'design' object computes the maximum moment and shear at 31 sections along the span of the beam for each load combination. It analyzes each span of the beam. Each section is then designed to determine the flexural steel requirements and the shear stirrups

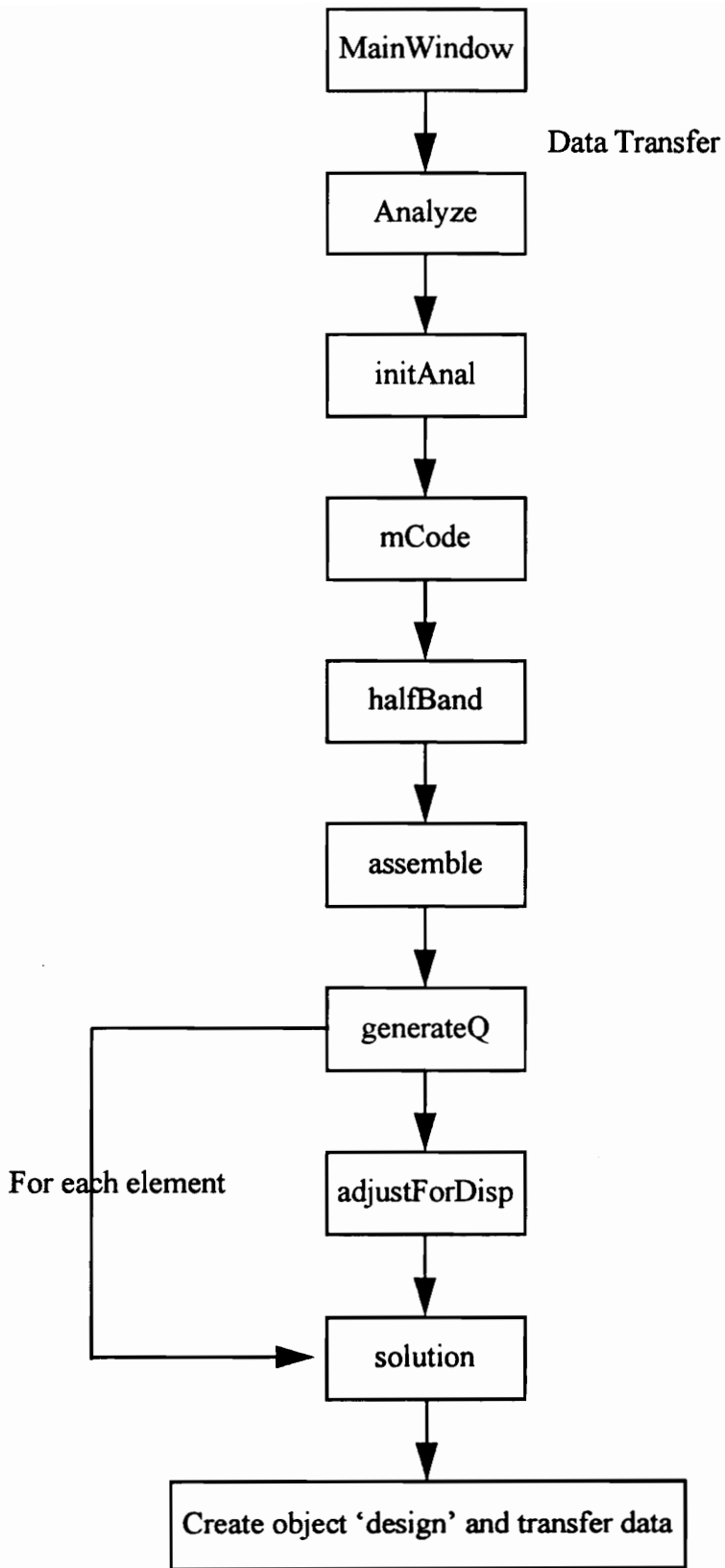


Fig. 4.35 Flow Chart of Object 'Analyze'

requirement. Fig. 4.36 shows a schematic diagram of the key methods of the class 'design' and their relationship.

The 'design' object creates an object 'simpObject', of the class 'Simple' and transfers the data to it. This class 'Simple' is again a sub class of the class 'ContBeam'. Fig. 4.22 shows the flowchart for the object 'simpObject'. As shown in the flowchart the 'simpObject' completes the design, displays the results of the final design on the screen, and writes the results to an output file.

The continuous beam application represents a real world object-oriented structural engineering application. The program has all of the features that are necessary for a typical Windows application. All elements of the program were developed using object oriented programming techniques. The four different objects, the mainWindow, Analyze, design, and the simpObject are representative of typical engineering applications. These objects involve elements of both the user interface, such as Windows and the dialog boxes, and the elements of structural analysis and design.

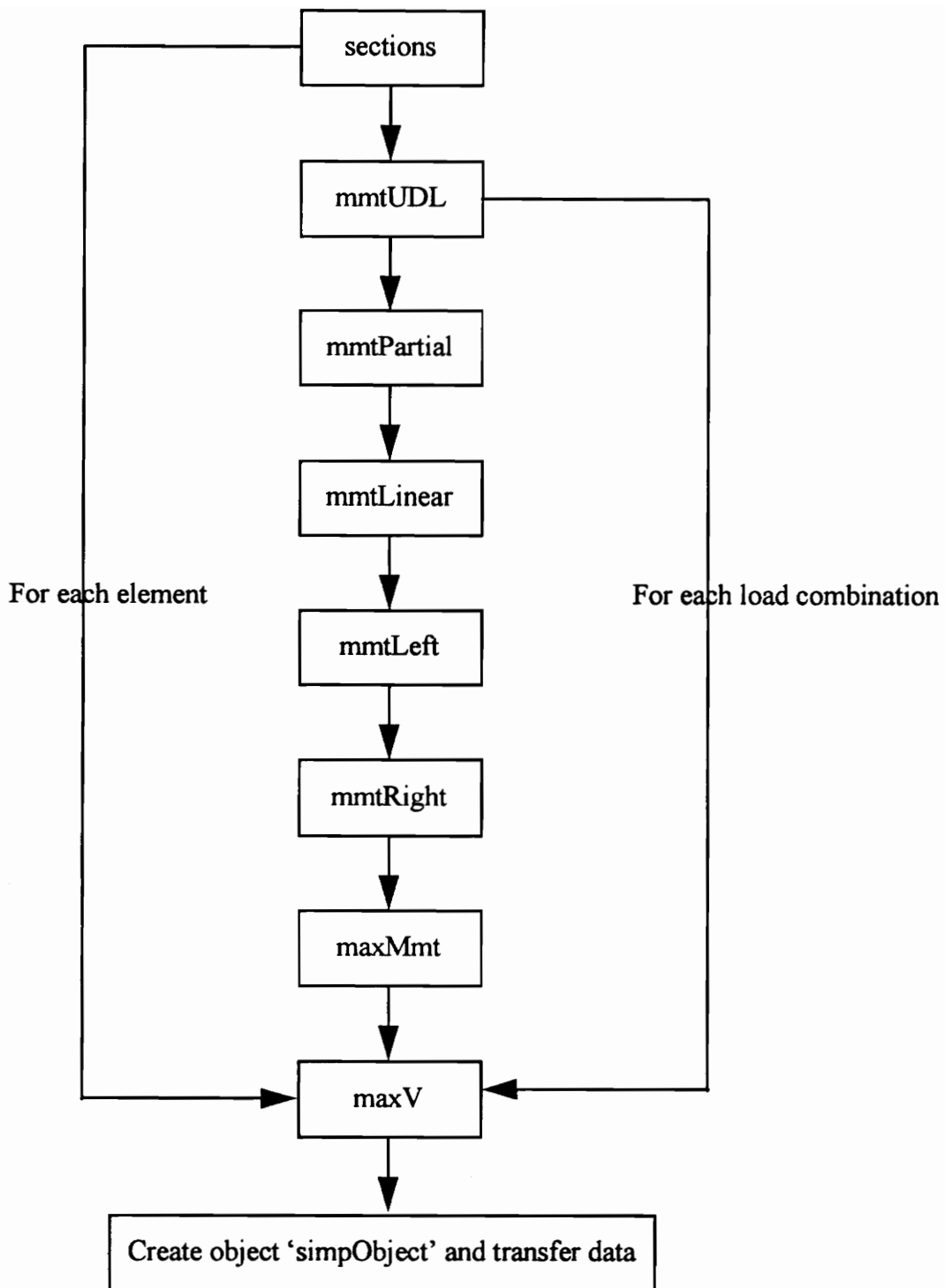


Fig. 4.36 Flow Chart of Object 'design'

Chapter 5

C++ Applications

5.1 Introduction

In this chapter the applications developed using C++ are discussed. In order to make a fair comparison of the two object oriented languages, C++ and ACTOR, two applications that were similar in scope and function to those written in Actor were developed using C++. The first application was a small program to compute the flexural capacity of a tee beam. The second application was the program to analyze and design a reinforced concrete continuous beam.

This chapter describes both these programs. It also discusses some of the similarities as well as some of the differences in these applications, as a result of using two different programming languages. Both applications were developed using the Borland C++ compiler (Version 3.0) for Windows with the Borland Object Windows Library of classes. The Object Windows Library (OWL) provides the necessary classes for developing Windows applications.

5.2 Flexural Capacity of Tee Beam Application

This program computes the flexural capacity of a tee beam in accordance with the ACI 318-89 specifications [33]. The input to the program consists of section properties of the beam, the area of the reinforcing steel and material properties. The program computes the flexural capacity of the section and displays the results and the input data on the screen. The display is modified each time the data are modified. An overview of this application was presented in Section 4.2.

As was the case with Actor, every Windows application written using Borland Object Windows Library has a sub class of the TApplication library class to start the

application. The TApplication class opens the main window of the application. This TApplication class has only one member function : 'InitMainWindow'. This function creates the main window of the application and calls the 'setAttr' method to set the location attributes of the main window. Fig. 5.1 shows the class structure of the Tee Beam program.

The main window of the application is defined by a sub class of the TApplication class, to be an object of the class 'TTeeWindow'. The 'TTeeWindow' class is derived from the OWL 'TWindow' class. The 'TWindow' class defines the basic features of the windows in the application. It can be used to create the main window or pop-up windows. The class definition of the 'TTeeWindow' class is shown in Fig. 5.2.

As can be seen from the Fig. 5.2, the data members of the class 'TTeeWindow' are declared as private while the functions are described as public. This is different from the instance variables used in Actor. In Actor, the instance variables are inherited by the sub classes, but the private data members of a class can't be inherited by a sub class. The functions, which are declared as public, can be accessed from any part of the program using an object of that class.

The 'TTeeWindow' class has two functions for obtaining input (see Fig. 5.2). The main window, an object of the class 'TTeeWindow', calls the functions 'CMGeometry' and 'CMMaterial' to obtain the section geometry and the material properties. The dialog box object 'matDlg' is created from the 'TDlg' class when the menu item for Material Properties is selected from the main menu. The 'TDlg' class is inherited from the OWL 'TDialog' class, which is a class that is specifically written for handling dialog boxes in Windows. Upon creation of the dialog object, data from the main windows object is transferred to the 'matDlg' object using arguments. The 'matDlg' object is created as an instance of the class 'TDlg'. This class has only two data members, fc and fy, both declared as private. When the 'OK' button of the dialog box is selected, the data from the dialog box is transferred back to the main window object.

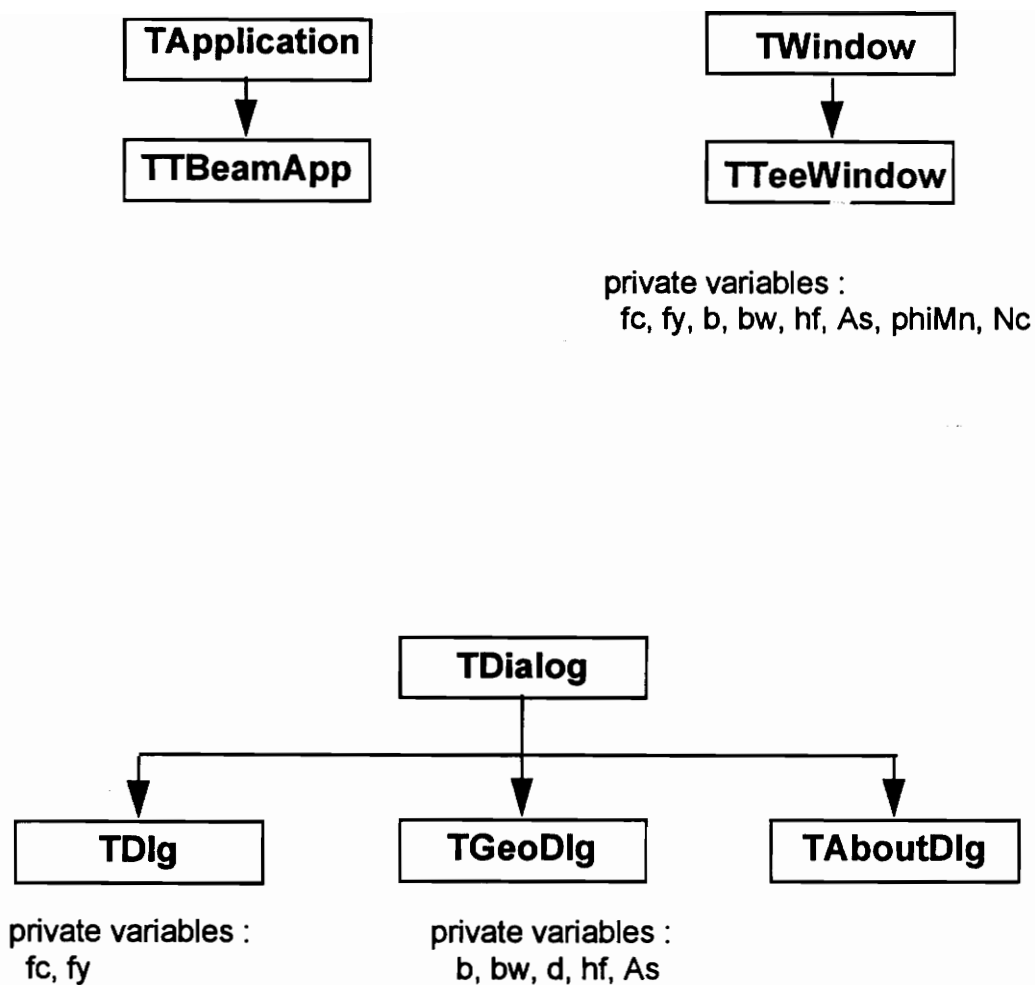


Fig. 5.1 Class Structure for the Tee Beam Application

```

// Main Window class definition
class TTeeWindow : public TWindow
{
private:
    float fc, fy, b, bw, d, hf, As, phiMn, Nc;

public :
    TTeeWindow(PTWindowsObject AParent, LPSTR ATitle);
    void setAttr();
    virtual void CMNew(RTMessage Msg) = [CM_FIRST + CM_New];
    virtual void CMPrint(RTMessage Msg) = [CM_FIRST + CM_Print];
    virtual void CMclose(RTMessage Msg) = [CM_FIRST + CM_Exit];
    virtual void CMGeometry(RTMessage Msg) = [CM_FIRST + CM_Geometry];
    virtual void CMMaterial(RTMessage Msg) = [CM_FIRST + CM_Material];
    virtual void CMAbout(RTMessage Msg) = [CM_FIRST + CM_About];
    void initiateData();
    void checkDuct();
    void capacity();
    virtual void Paint(HDC PaintDC, PAINTSTRUCT _FAR & PaintInfo);
};

```

Fig. 5.2 Class Definition for 'TTeeWindow' class

The geometric properties for the beam section are entered through an object 'geoDlg' of the class 'TGeoDlg'. The 'TGeoDlg' is also derived from the OWL 'TDialog' class. This class has the variables of b, bw, d, hf and As, declared as private data members. There are no public data members. This class also has member functions for closing the dialog box when the 'Ok' or 'Cancel' button is pressed. When the dialog box is closed using the 'OK' button, data is transferred to the main window object. When the 'OK' button of the dialog box is pressed, each data member is checked. If any of the values entered are negative, then a message is displayed indicating that values entered are invalid.

On creation of the dialog object, but before displaying the dialog box, geometric data from the main window are copied to the 'geoDlg' object. When the geometric data are entered or modified, these are assigned to the data members of the 'geoDlg' object. The data are then transferred to the main window object when the 'geoDlg' object is closed with the 'OK' button.

For the analysis of the section, the main window object uses the functions 'checkDuct' and 'capacity'. The 'checkDuct' function performs the ductility check on the section and computes the effective steel area for the section. The 'capacity' function computes the moment capacity. A windows paint message, 'WM_PAINT' message, is then sent to the main window to update the results, which are displayed in the main window. The display is modified each time the data is modified.

This application program is a simple tool for computing the flexural capacity of tee sections. The best feature of this application is its ability to modify the section capacity each time the section or material properties are changed. Although this program is relatively simple to develop using object oriented programming techniques it provides a good example of using the Borland Object Windows Library to develop Windows applications.

5.3 Analysis and Design of R. C. Continuous Beam

The second application developed using C++ was a program to analyze and design a reinforced concrete continuous beam. While developing applications in ACTOR, a program for the design of a simply supported beam followed the development of the tee beam application. Although the simply supported beam design application was thought to be useful, it eventually turned out to be redundant after the development of the continuous beam design program since the continuous beam program can also handle a one span simply supported beam. Thus, the program for the design of a simply supported reinforced concrete beam was not developed in C++.

The continuous beam application can handle up to twelve spans with up to ten load combinations and ten load cases. The following types of loads can be applied on the beam:

1. Uniformly distributed load along the entire span
2. Partially distributed load
3. Concentrated loads
4. Linearly varying loads.

Each span can have up to ten concentrated loads, and five each of any other type of load. The program analyzes the structure for each load combination. The analysis is done using the stiffness matrix method using Holzer's approach [27]. Each span of the beam is divided into 30 equal parts for moment and shear computations. After analyzing the structure for each load combination, the maximum moment and shear is computed at each section. Then the section requirements are determined. After the section has been chosen, the flexural steel requirements are determined for all 31 sections along each span. For the selected stirrup size the spacing for U shaped stirrups is also computed at each section. The details of this application are given in Section 4.6.

Fig. 5.3 shows the class structure of the continuous beam application. The subclass 'ConApp' starts the application by creating the 'MainWindow' as an object of the

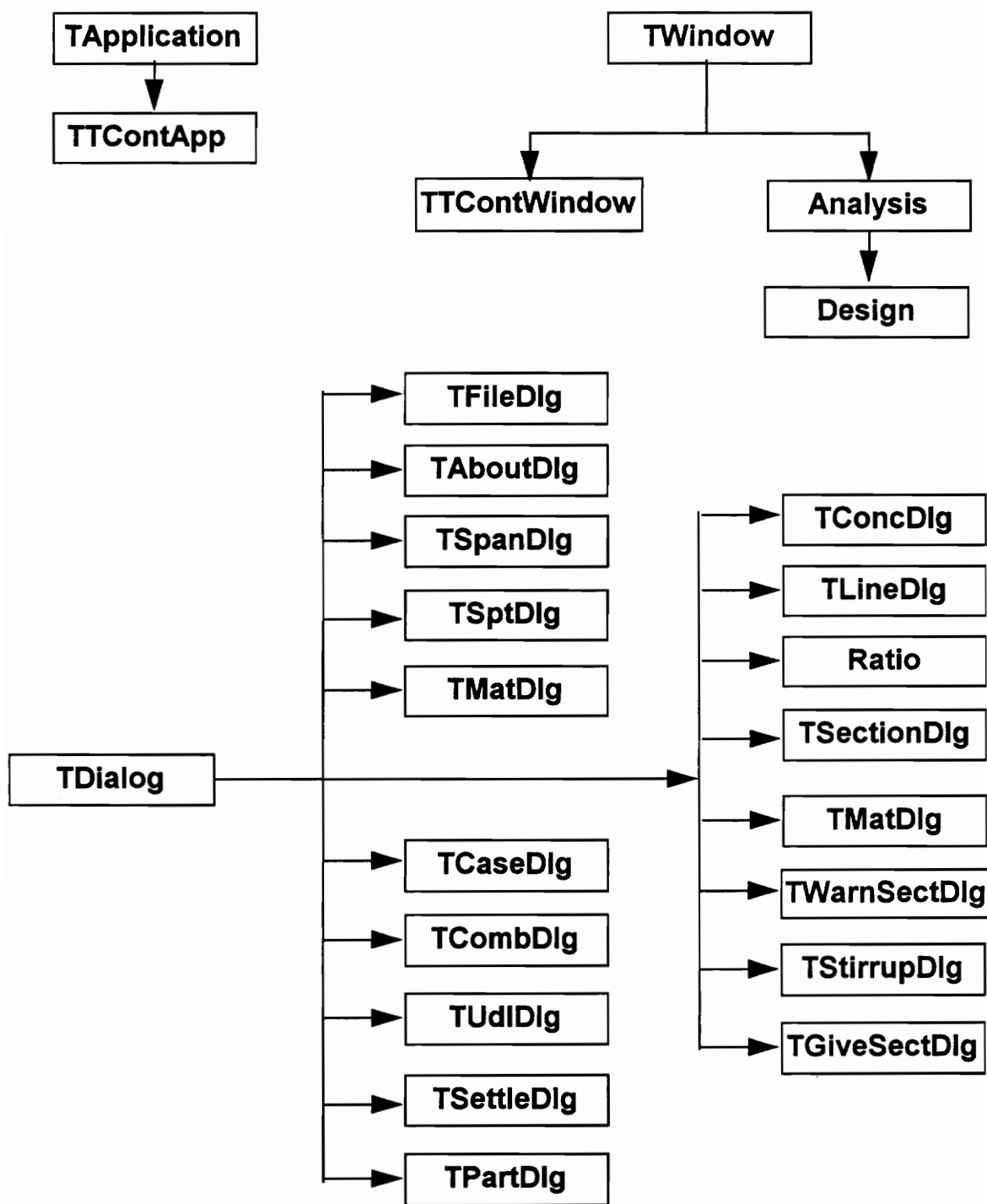


Fig. 5.3 Class Structure for the Continuous Beam Application

class 'TTContWindow'. The 'TTContWindow' class has the necessary data for both the analysis and design. These data members are declared as protected in the 'TTContWindow' class definition. The class definition also includes the declaration of all functions as public members. The 'TTContWindow' class has functions for obtaining input data for the analysis and the design. Appendix A shows the class definition for all of the classes of the continuous beam application.

When the corresponding menu item for entering data is selected from the main window, a dialog box is created and displayed. For example, when the 'Load Cases ...' menu item is selected the 'CMCases' function of the 'TTContWindow' class is invoked. This function creates and displays a dialog object of the class 'TCaseDlg'. This dialog box object has its own set of private data for storing data on load cases. When the dialog box is first created, load case data from the main window object is transferred to the dialog box object. The dialog box object copies the contents of the dialog box data buffer into its private data. Then it allows the user to manipulate this copy of the data. The class definition of the 'TCaseDlg' dialog class is shown in Fig. 5.4. The 'setCase' function copies the information on load cases from the dialog data buffer. The 'caseDone' function closes the dialog box when the 'Done' button of the dialog box is selected. It also copies the load case data to the data buffer. The 'caseNext' and 'casePrev' functions access the next and the previous load case data when the 'Next' and 'Previous' buttons of the dialog box are clicked. When the dialog box is closed, the data from the data buffer is copied to the main window's data. The dialog box object, and its data buffer, exists only during the execution of the function 'loadCase' of the 'TTContWindow' class.

The 'Analysis' class, a sub class of the 'TWindow' class, performs the stiffness analysis of the structure. When the 'Analyze' menu of the main window is selected, an object 'analyze' of the class 'Analysis' is created. This class has protected data members for all the loads, spans and the material properties. In addition to this protected data it also has private data for the global stiffness matrix and the moments. Appendix A shows the


```

_CLASSDEF(TCaseDlg)

// class definition for the load case dialog
class TCaseDlg : public TDialog
{
private :
    TEdit *Edit1;          //Edit box for load case
    TStatic *Static1;      // static control for name
    char loadCase[10][70];
    int count, NEL;

public :
    TCaseDlg(PtWindows Object, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
        + WM_INITDIALOG];
    void setCase();
    void caseDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void caseNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void casePrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
};

```

Fig. 5.4 Class definition for the class 'TCaseDlg'

class definition for this class. The protected data members are inherited by the subclass 'Design'. The private data members are only available to the functions of the class 'Analysis'.

The 'analyze' object, when first created, uses the setData family of member functions to get the data from the main window which is an object of the TTContWindow class, via the data buffer. Once the data is transferred, the 'analyze' object starts the analysis process by executing the function 'mCode'. This function, along with the 'oneEleMCode' and 'mCode2' functions, computes the M code matrix for the given beam configuration. The M code matrix defines the compatibility conditions for the beam configuration [27]. Once the M code matrix is generated, the half band width of the global stiffness matrix is determined. Function 'halfBand' does this computation. Then the function 'assemble' assembles the global stiffness matrix. The global stiffness matrix is stored using skyline storage approach [34]. The 'generateQ' function computes the joint load vector for each load combination. The stiffness matrix is then modified for the prescribed displacements by the 'adjustForDisp' function. This function precedes the solution of the equation $kq = Q$ [27]. The Gaussian Elimination procedure starts with the conversion of the stiffness matrix to an upper triangular matrix [34]. The joint displacements are computed by back substitution of the upper triangular stiffness matrix. The support moments are computed by the 'eleF' function. The 'solution' function creates a 'design' object from the class 'Design'. The data and the results of the analysis are transferred to this 'design' object through the data buffer.

The 'Design' class is a subclass of the 'Analysis' class. Thus, the 'Design' class inherits all of the protected data members of the 'Analysis' class. The 'Design' class also has the section properties and stirrup information stored as its private data members. All of the functions are declared as public. These functions compute the flexural and shear reinforcement requirements. The design starts with one of three choices :

1. Design for Given Section
2. Design for Maximum Moment
3. Final Design.

If the 'Design for maximum moment' option is selected, the design procedure begins by determining the maximum moment in each span of the beam. The function 'maxMmt' performs this computation. Then the user is prompted for the steel ratio to be used in the design. A dialog box of the class 'Ratio' obtains the user input for the steel ratio. A range of sections is then determined for the given steel ratio and the computed maximum moment that satisfy the requirements for flexure. The function 'trialSection' computes the required section and obtains the user's choice of the section through a dialog box of the class 'TSectionDlg'. This section is then checked for adequate flexural capacity through the 'chkCap' function. The 'chkCap' function uses the 'ductCheck' function for the ductility check. A message is displayed if the section needs to be doubly reinforced. After the capacity check, the flexural steel requirement is computed by the 'afterChkCap' function. The user is then asked for the stirrup bar number. The required stirrup spacing for the trial section is then determined. The results are displayed in the main window by sending a 'WM_PAINT' message to the main window object.

The 'Design for Given Section' option performs the flexural and shear design for a section given by the user. Since the section dimensions are known, only the functions for the flexural and shear design are executed.

The 'Final Design' can only be executed after the beam section has been selected. The section is designed in a manner similar to the 'Design for Given Section' option. After the final design has been done, an object of the class 'readData' is created and the output data file is written.

The 'readData' class is derived exclusively for dealing with the file handling functions. It has two distinct types of functions : one for reading a data file, and the other

for writing the output file. Private data members of the class are used to store the data from the main window.

This chapter described the application programs developed using C++. The Tee beam program was a small but useful application. It provides a tool similar to a spread sheet for computing the flexure capacity of a tee beam with given dimensions and reinforcement. The continuous beam design application program is a good example of a practical situation analysis and design application and provides a basis for comparison with a similar application developed using Actor. The program can easily be modified to perform the analysis and design of plane frames. In the next chapter results of this study are presented.

Chapter 6

Results

In this chapter a comparison of the Actor and C++ programming languages is presented. Issues that are important for developing structural engineering applications, using object oriented programming are also addressed.

6.1 Comparison of Actor and C++

6.1.1 Syntax

The syntax for Actor is a mixture of Pascal and C. For example, the use of curly brackets to enclose methods and the use of a semicolon at the end of each statement, are similar to C. The use of 'EndIf' after 'If' and the use of ':=' for assignment are similar to Pascal.

```
A := B*C;
```

Thus, in each statement, the Actor syntax resembles a combination of the C and Pascal syntax.

Although the syntax of the Actor language is somewhat similar to C and Pascal, considerable effort is required in learning the language. This is due to the fact that Actor is pure object oriented programming language and many of the features of the language are unfamiliar to most programmers.

Borland C++ is an extension to the popular programming language C. The syntax of C++ is very similar to the C programming language. This makes the transition from C to C++ a very easy one for someone who is familiar with C. C++ does have extensions, especially those for developing classes, that are new and so some effort is required in learning these new features.

It should be noted, however, that the development of an object oriented application requires a totally new kind of approach to program development since it is unlike a procedural program. Considerable experience is needed to master the skills needed to create meaningful classes and in structuring the program to take advantage of the features of object oriented programming.

6.1.2 Environments

The Actor environment is probably one of the best programming environments. With a separate window for the Class Browser and the Attribute Browser, it is really very different from conventional programming environments. With only one method displayed at a time, and no space to declare global variables, it is a complete object oriented environment. All instance and class variables are declared at the class definition level, above the methods, making them available to all methods of a class. From a programmer's perspective it is an excellent programming environment , though a bit slow and difficult to learn.

The Borland C++ Windows environment is an integrated development environment with the text editor, compiler and debugger options all available from the same Windows environment. This makes application development very easy. It is also possible to write C programs using the Borland C++ compiler. This is very important since it makes it possible to reuse previously developed functions.

Since run time binding is the norm in Actor, this results in considerably longer execution times for all applications. For example, in the continuous beam application the analysis of a beam took seven seconds on a 80386 IBM PC with a 33 MHz processor. The corresponding C++ application took less than two seconds. Also, the Actor compiler is very slow. In comparison, the C++ compiler is extremely fast. Even when run time binding was performed in C++ for the continuous beam application, it was still sufficiently faster than the corresponding Actor application.

In Actor, every method is compiled before saving. This makes it possible to test the application during its development. However, the Actor compiler is very slow in compilation and since the Actor compiler is not an incremental compiler, whenever a minor change is made in a method the compiler has to recompile the entire method. Also, if variables in a class definition are modified then all methods in that class and its sub classes are recompiled. The C++ compiler is extremely fast and compilation takes only a few seconds. The C++ compiler also takes advantage of precompiled header files which speeds up the compilation process considerably.

Actor provides a much easier way to allocate dynamic memory as compared to C++. In Actor, any variable is a type free variable and its type need not be defined until the time of assignment. This means that a variable can hold any value, a real, an integer, a string, and a particular variable can represent any of the data types in the program. This feature made it possible to dynamically allocate memory for the stiffness matrix and displacement and load vectors in the continuous beam application. Although it's possible to allocate memory dynamically in C++ it was done indirectly using pointers.

The Actor documentation leaves a lot to be desired. The programming manuals do not provide a comprehensive list of all the classes and their methods. Although the Class Browser shows a list of all the classes and the corresponding methods, there is no easy way to determine what each method does. In order to determine the purpose of a particular method it was necessary to open the method and read the comments in the method.

The Actor Debugger is a better tool as compared with the Borland C++ Debugger. The debugger allows backward tracing and it is possible to modify methods. The application can be started from that particular method with all the variables retaining their data values. This saves considerable time since it is not necessary to start the application from the beginning. The Borland C++ debugger is not as powerful as the Actor Debugger. Also, the Borland C++ debugger is a DOS application and not a Windows

application, thus making it necessary to switch between DOS and Windows during a debugging session.

One of the most difficult aspects of the Actor environment was the effort required in creating an executable file. Actor compiles all the methods while saving them. This makes it possible to test the application during development. When the application is completely debugged, creating a stand-alone executable file requires considerable effort in compiling and linking the source and resource files. In Borland C++, once a project has been created, compiling, linking and creating executable file is a very simple task. All that is required is to select the 'Make' option from the 'Run' menu.

Actor is built on the Windows environment. It is not possible to develop DOS applications using Actor. The Actor class structure is equipped with the necessary classes for developing Windows applications. In Borland C++, the Borland Object Windows Library of classes was used to develop Windows applications. Although it is possible to develop Windows programs in C++ without using OWL, the process is extremely tedious and is not recommended.

The Borland C++ (Version 3.0) compiler provided the comforts of a standard, well established compiler. The elegant windows environment made it much easier to develop applications as compared to Actor programming. By the time the applications were developed in C++, the object oriented programming, methods and classes were fully understood. This, along with the advantage of knowing C, made application development considerably faster in C++ than in Actor. In view of the above discussion, it is author's opinion that the Borland C++ is a better platform for application development than Actor. A comparison of Actor and C++ is presented in the Tables 6.1.

6.2 Applicability of Object Oriented Programming in Structural Engineering

Miller [7] has suggested three important criteria: software development, maintainability and usability for structural engineering software. In this section these

Table 6.1 Comparison of Actor and C++

Feature	Actor	Borland C++ for Windows
Syntax	Resembles C and Pascal.	Resembles C.
Environment	Complete integrated development environment with a good Browser, Debugger and Inspector.	Complete integrated development environment with text editor, compiler, linker and debugger.
Object Oriented Programming	Pure object oriented language. Good tool for learning OOP.	Hybrid nature allows programmers to make use of previously written C code.
Application Development and Testing	Application can be tested while in the development stage.	An executable file has to be created for testing.
Execution	Run time binding and message passing make it slow in execution.	C compiler generates fast and efficient executable files.
Debugging	Powerful Debugger and Inspector make debugging very easy.	Debugger is provided but is not as powerful as the Actor Debugger.
Creation of executable file	Several steps have to be performed to compile and link the source file and resources.	Excellent integrated development environment for compiling and linking all source files and resources.
Editor	Buffer size for any method is only 112 lines making it impossible to have methods having more than 112 lines.	No practical limitations on size of source files. Excellent text editor.
Windows Application Development	Since the entire environment is built on Windows it is easy to develop Windows applications.	Object Windows Class Library is used to make it easier to develop Windows applications.

issues will be discussed focusing on the applications developed during this study. The discussion focuses on the advantages and disadvantages of object oriented programming while developing structural engineering applications.

The basic advantage of object oriented programming is increased modularity. Software developers tend to develop programs into small modules to achieve ease of software development and maintenance. In object oriented programming, since each object is an independent module, it provides maximum modularity. Since each object contains data and functions in one entity, it is very easy to develop a program as a series of objects. This technique was evident in the development of the continuous beam application. In the continuous beam application first the main window object was created to assist data entry. Once the main window object was created, the program had the basic user interface for the data entry completely debugged and then an object of the class 'Analysis' was created to perform the analysis. The only link between the 'MainWindow' object and the 'Analyze' object was the data transfer functions. Thus, during the development of the 'Analyze' object changes in the 'Analyze' object did not affect the 'MainWindow' object. This increased modularity is a significant advantage of object oriented programming during the software development and maintenance stages over the conventional programming paradigms.

Another feature of object oriented programming that has considerable potential for structural engineering applications is inheritance. In the continuous beam application developed using Borland C++, when the class 'Analysis' was developed, all input variables were declared as protected data members of the class. These data members included span data, load data and material properties data. The data members for the analysis part of the class, such as, stiffness matrix and load vector were declared as private members. Thus, when the 'design' class was declared as a sub-class of the 'Analysis' class, all protected data members in the 'Analysis' class were also available to objects of the class 'design'. This saved the effort required in the redeclaration of the data members for geometry, load and

material properties data in the class 'design'. This feature of inheritance can be quite useful if the same application is modified to design plane frames. In this case, it is possible to develop another sub-class from the 'Analysis' class to design column elements of the frame while the existing class can design beam elements. Another class can be developed from the column design class and the beam design class for the design of beam-columns. Thus, the objects 'Analyze' and 'design' can be used as part of a frame analysis and design program which would save considerable programming effort. The ability to reuse previously developed and tested objects and to derive new objects from these objects is a significant advantage since it can considerably reduce program development time.

Another advantage of the object oriented programming is that objects depict real world situations. For example, for the continuous beam application development of separate objects for the analysis and design parallels those in the real world. When a beam is designed, it is first analyzed and then designed. This function based classification can be extended further for a frame in which after the analysis is performed, a beam object will design the beams while column objects will design columns in the structure.

This study proves that object oriented programming does indeed make software development, maintenance and usability of the code easier than conventional programs. It should be noted that these advantages of object oriented programming do not come without a few overheads. First, it is necessary for structural engineers to learn object oriented programming. A conventional programmer may find it difficult to understand the world of classes, objects and functions. The transition from conventional programming to object oriented programming is a difficult one. Object oriented programming places additional responsibility on the programmer to plan the objects and functions in advance. However, this additional effort can result in a considerable saving in the future application development.

The advantages of object oriented programming are more evident for large applications. In smaller applications the time spent in the planning and development of

objects may not be justified. To conclude, it may be stated that object oriented programming definitely increases program modularity which leads to ease in software development and maintenance. Also, the inheritance feature makes it possible to reuse previously developed and tested objects, hence resulting in considerable savings in the effort required in developing new applications.

Chapter 7

Summary and Conclusions

In this chapter a summary of the study and the major conclusions are presented. The chapter also discusses the success of the study against its slated objectives. The advantages of object oriented programming, as stated in the chapter 2, and as experienced during the course of application development are also summarized.

7.1 Summary

The main objectives of this study were to apply object oriented programming technology to structural engineering and to study its advantages for structural engineering applications. The study was based on two object oriented programming languages, Actor and Borland C++. Both languages were used to develop two programs: Flexural Capacity of Reinforced Concrete Tee Section and Analysis and Design of Reinforced Concrete Continuous Beam. A third application, Design of Simply Supported Reinforced Concrete Beam, was also developed in Actor. The Continuous Beam application is an example of a practical structural engineering application and provided an opportunity to evaluate the use of object oriented programming for structural engineering. Since programming styles and environments of Actor and C++ differed considerably, a comparative study of the two languages was also performed.

7.2 Conclusions

The development of two identical programs in Actor and C++ helped assess the benefits and weaknesses of each of these languages. While Actor proved to be a superb teacher of object oriented programming, Borland C++ was a better programming tool.

The convenience of embedding conventional C functions in C++ provided a smoother transition for procedural programming to object oriented programming. The Actor environment, although better than Borland C++ environment, lacked the versatility of the later. It took special effort to create a stand-alone Windows application in Actor. In the Borland C++ integrated development environment it was a matter of simply selecting the 'build' menu item to create the executable program. The Actor compiler was considerably slower than the Borland C++ compiler. Based on the experience of programming in both Actor and Borland C++ it is concluded that the Borland C++ is the better environment for application development.

From a structural engineering view point, it was found that the use of object oriented programming resulted in modular programs. This was very helpful in the development and debugging process. Also, the resemblance of objects to real world items such as analysis and design made the program easier to understand. The inheritance feature of object oriented programming was of importance in the continuous beam program. It can be used further to develop a program to design plane frames. Also, it may be mentioned that object oriented programming needs considerable thought during the program development and planning stage. The benefits of object oriented programming are possible only if the objects are well planned. It also takes considerable effort to learn object oriented programming and to develop the skills and experience needed to take advantage of the technology.

References

1. E. H. Tyugu, "Object Oriented Programming", *Programming and Computer Software*, Vol. 18, No. 8, 1991.
2. W. Schubert and H. Jungklaussen, "Characteristics of Programming Languages and Trends in Development", *Programming and Computer Software*, Vol. 16, No. 5, 1991.
3. J. Micallef, "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object Oriented Programming*, April/May 1988.
4. W. J. Cook, "Object Oriented Programming Versus Abstract Data Types", *Foundations of Object Oriented Languages, Lecture notes in computer science*, Vol. 489, REX School/Workshop, May/June 1990.
5. J. W. Hopkins, "Objective C - Object Oriented Programming : The Next Step Up", *Journal of Object Oriented Programming*, May/June 1990.
6. H. Sallie, M. Humbrey and J. Lewis, "Evaluation of Maintainability Of Object Oriented Software", *Proceedings of the IEEE Region 10 Conference*, September 24-27, Hong Kong, 1990.
7. G. R. Miller, "What Object Oriented Programming Can Mean For Structural Engineers", *Electronic Computation, Proceedings of the Tenth Conference, Indianapolis, 1991*.
8. A. S. Watson and S. H. Chan, "A PROLOG-based Object Oriented Engineering DBMS", *Computers & Structures*, Vol. 40, No. 1, 1991.
9. G. R. Miller, "An Object Oriented Approach to Structural Analysis and Design", *Computers & Structures*, Vol. 40, No. 1, 1991.
10. G. R. Miller, "A LISP-based Object Oriented Approach to Structural Analysis", *Engineering with Computers*, Vol. 4, No. 4, 1988.

11. G. R. Miller, "Object Oriented Concurrent Structural Analysis", *Computing in Civil Engineering, Proceedings of the Sixth Conference*, September 11-13, Atlanta, 1989.
12. G. H. Powell, G. A. Abdalla and R. Sause, "Object-Oriented Programming Knowledge Representation : Cute Things and Caveats", *Computing in Civil engineering, Proceedings of the Sixth Conference*, September 11-13, Atlanta, 1989.
13. G. H. Powell and Rajiv Bhateja, "Data Base Design for Computer Integrated Structural Engineering", *Engineering with Computers*, Vol. 4, No. 3, 1988.
14. H. N. Al-Nashif and G. H. Powell, "An Object-Oriented Algorithm for Automated Modeling of Frame Structures: Stiffness Modeling", *Engineering with Computers*, Vol. 7, No. 2, 1991.
15. G. H. Powell and G. A. Abdalla, F. Filippou, "An Object Oriented Approach to Computer Aided Reinforced Concrete Design", *Third International Conference on Computing in Civil Engineering*, August 10-12, Van Couver, B.C., Canada, 1988.
16. H. Adeli and G. Yu, "Computer Aided Design of Structures", *Microcomputers in Civil Engineering*, Vol. 6, No. 3, 1991.
17. P. Remy, B. Devloo and J. S. Rodrigues Alves Filho, "An Object Oriented Approach to Finite Element Programming (Phase I): A System Independent Windowing Environment for Developing Interactive Scientific Programs", *Advances in Engineering Software*, Vol. 14, No. 1, 1992.
18. M. J. Rice, "An Object Oriented Approach to Freeway Analysis and Design", *Third International Conference on Computing in Civil Engineering*, August 10-12, Van Couver, B.C., Canada 1988.
19. A. A. Oloufa, "Modeling Operational Activities in Object Oriented Simulation", *Journal of Computing in Civil Engineering*, Vol. 7, No. 1, 1993.
20. J. H. Garrett, "An Object Oriented Representation of Design Standards", *Computing in Civil Engineering, Proceedings of the Sixth Conference*, September 11-13, Atlanta, 1989.

21. G. H. Powell and G. A. Abdalla, "Object Management in An Integrated Design System", *Computing in Civil Engineering*, Proceedings of the Sixth Conference, September 11-13, Atlanta, 1989.
22. W. Rotzheim, "Programming Windows with Borland C++", 1992.
23. K. Christian, "The Microsoft guide to C++ Programming", *Microsoft Press*, 1992.
24. B. Ezzel, "Turbo C++ Programming", *Addison-Wesley Publishing Company, Inc.*, 1990.
25. E. R. Tello, "Object Oriented Programming for Windows", *John Willey & Sons, Inc.*, 1991.
26. J. W. McCord, "Developing Windows Applications with Borland C++3", *SAMS*, 1991.
27. S. M. Holzer, "Computer Methods of Structures", *ELSEVIER*, 1985.
28. L. Atkinson and M. Atkinson, "Using Borland C++", *QUE Corporation*, 1991.
29. International Data Corporation, "White Paper, Object technology : A Key Software Technology for the 90's", *Computerworld*, 1992.
30. "Actor Programming Manual", *Whitewater Group*, Evanston, Illinois, 1991.
31. "Borland C++ Programming Manual", Borland Inc., Scott Valley, CA, 1992.
32. R. Lafore, "Object Oriented Programming in Turbo C++", *Waite Group Press*, Mill Valley, CA, 1991.
33. American Concrete Institute, "Building Code Requirements for Reinforced Concrete (ACI 318-89) and Commentary - ACI 318R-89, Detroit, Michigan, 1990.
34. K. J. Bathe and E. L. Wilson, "Numerical Methods in Finite Element Analysis", *Prentice-Hall, Englewood Cliffs, NJ*, 1976.

Appendix A

Class Definitions for Continuous Beam Application in C++

```

/*****
    Application class for the Continuous Beam Application
*****/

_CLASSDEF(TTContApp);

// Application class definition
class TTContApp: public TApplication
{
public :
    TTContApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
              LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
              hInstance, hPrevInstance, lpCmdLine, nCmdShow)
    {};
    virtual void InitMainWindow(); // Function to open main window
};

/*****
Variables:
    span          Span of each element of continuous beam
    I              Moment of inertia for each span of the continuous beam
    fc             Compressive strength of concrete
    fy             Yield strength of steel
    loadCase       Load cases acting on the beam
    loadComb        Name of load combinations
    factor          Load factors
    w              Intensity of uniformly distributed load
    udlType         Load case for the UDL
    partW           Intensity of partially distributed load
    partA           Distance of partial load from left support
    partC           Spread of the partial load
    partType        Type of partially distributed load
    P              Concentrated load
    concA           Distance of concentrated load from left support
    concType        Type of concentrated load
    lineW1          Intensity of linearly varying load at the left end
    lineW2          Intensity of linearly varying load at the right end
    lineC           Spread of linearly varying load
    lineType        Load case for linearly varying load
    support         Support condition for the end supports
    jtDisp          Support settlement
    Q              Load vactor
    M              Structure compatibility matrix
    J              matrix for joint conditions
    mBand           Variable band matrix
    DOF             Degrees of Freedom
    SS             Stiffness matrix
    EMod            Modulus of elasticity
    mmt            Moment along span
    trialB          Width of trial section of beam
    trialD          Depth of trial section of beam
*****/
```

```

V          Shear along span
maxMmt     Maximum moment
maxV       Maximum shear
x          Distance of section from left support
AsRqd      Required tension steel
AslRqd     Required compression steel
Beta        $\beta$  value as per ACI
Mmt        Moment at the design section
ratio      Design steel ratio
b          Design width of the beam
d          Design depth of the beam
stBar      Stirrup bar size
s          Design spacing of stirrups
*****/

_CLASSDEF(Analysis)

// Analysis class definition
class Analysis : public TWindow
{
protected :
    float span[12], I[12], fc, fy, factor[10][10], w[12][5], tDisp[13],
        *Q;
    float partW[12][5], partA[12][5], partC[12][5], P[12][10],
        concA[12][10];
    float lineW1[12][5], lineW2[12][5], lineA[12][5], lineC[12][5];
    float leftMmt[10][12], rightMmt[10][12];
    int NEL, combination, support[2], udlType[12][5], partType[12][5],
        concType[12][10], lineType[12][5];

private :
    int **M, **J, mBand, DOF, *MAXA; // mBand = half band width
    float *SS, EMod, **mmt;

public :
    Analysis (PTWindowsObject AParent = NULL, LPSTR Title = NULL)
        : TWindow(AParent, Title){};

/* The setData functions are used to transfer data from main window
to objects of analysis object */
void setData1();
void setData2();
void setData3();
void setData4();

void initData(); // Initialization of member variables
void initAnal(); // Function to start the analysis
void mCode();    // Computation of M code
void oneElemCode(); // Computation of M code for beam with one
// span
void mCode2();   // Computation of M code
void jCode();    // Computation of J code
void halfBand(); // Computation of half band width of the
// stiffness matrix
void assemble(); // Assembly of Global stiffness matrix
// using skyline storage
void computeMaxa(); // Computing MAXA matrix
void generateQ();   // Computing load vector

```

```

float udlFixed(int, int); // Fixed end moments due to UDL
float partFixed(int, int); // Fixed end moment due to partial load
float concFixed(int, int); // Fixed end moment due to concentrated
                             // load

/* The following eight methods compute the fixed end moments due to
   the linearly varying loads */

float linearFixed(int, int); // Fixed end moment due to linearly
                             // varied load
float linear1(int, int, int);
float linear2(int, int, int);
float linear11(float, float, float, float, float);
float linear12(float, float, float, float, float);
float linear13(float, float, float, float, float);
float linear21(float, float, float, float, float);
float linear22(float, float, float, float, float);
float linear23(float, float, float, float, float);

void adjustForDisp(); // Adjusting for joint settlements
void eleF();          // Computing element forces

/* Solution of the stiffness equation */
void forSub1();        // Forward substitution
void fact1();          // Factorizing the stiffness matrix
void solution();       // Solution function

float fixedMmt(int, int); // Fixed moments due to continuity

void initDesign();      // Creates the design object and
                        // starts design
void afterRatio();

};

_CLASSDEF (Design)

class Design : public Analysis
{
private :
    float *trialB, *trialD, *mmt, *V, maxMmt[12][31], **maxV, *x,
          **AsRqd, **AsLRqd;
    float Beta, Mmt, ratio, b, d;
    int stBar, **s;

public :
    // Constructor for the design object
    Design(PTWindowsObject AParent = NULL, LPSTR Title = NULL)
        : Analysis(AParent, Title){ b= d= 0;};

    void afterData ();        // Function to start design after data
                             // transfer
    void spanDesign();        // Function to start design of each span
    void givenDesign();       // Function to start design if the section
                             // is given by the user
    void finalDesign();       // Starting final design after section is
                             // fixed
    void sections(int);       // Computing sectional requirement

/*****

```

The following seven functions compute the moment along each span of the beam considering each span as a simply supported beam subjected to support moments.

*****/

```
void mmtUdl(int);      // Moment due to distributed load
void mmtPartial(int); // Moment due to partial distributed load
void mmtPoint(int);   // Moment due to concentrated loads
void mmtLinear(int);  // Moment due to linearly varying load
```

```
// Moment due to triangular loads
void triangle(float, float, float, float, float, int);
```

```
void mmtLeft(int);
void mmtRight(int);
```

```
// Functions to compute maximum moment and shear along each span
void maxMmt1(int);
void maxV1(int);
```

Design functions: Design as per ACI 318-89

*****/

```
void betaCal();      // Computation of  $\beta$  value
void absMaxMmt();    // Computation of maximum moment
void aftRatio();     // Function to start design after taking
                    // input of steel ratio
void trialSection(); // Computing sectional requirement
void afterSection(); // After taking user's choice for section
void chkCap();       // Computing flexural capacity of section
int afterChkCap();   // After checking capacity starting shear
                    // design
int ductCheck();     // Performing ductility check
void deleteSteel();  // Function to free the computer memory
void deleteShear();  // Function to free the computer memory used
                    // by shear design variables
void shearDesign();  // Function to design section for shear
void outFile1(char[]); // Writing output file
```

```
};
```

```
_CLASSDEF(TTContWindow);
```

// Main window class definition

```
class TTContWindow : public TWindow
```

```
{
```

```
protected :
```

```
float span[12], I[12], fc, fy, factor[10][10], w[12][5],
    jtDisp[13];
float partW[12][5], partA[12][5], partC[12][5], P[12][10],
    concA[12][10];
float lineW1[12][5], lineW2[12][5], lineA[12][5], lineC[12][5];
int NEL, support[2], udlType[12][5], partType[12][5],
    concType[12][10], lineType[12][5];
char loadCase[10][70], loadComb[10][70];
```

```
public :
```

```
TTContWindow(PWindowsObject AParent, LPSTR ATitle);
```

```

void setAttr();
void initiateData();
virtual void Paint (HDC PaintDC, PAINTSTRUCT _FAR & PaintInfo);

// Following functions handle selection of a menu choice from the
// 'File' menu of the main window
virtual void CMNew(RTMessage Msg) = [CM_FIRST + CM_New];
virtual void CMOpen(RTMessage Msg) = [CM_FIRST + CM_Open];
virtual void CMSave(RTMessage Msg) = [CM_FIRST + CM_Save];
virtual void CMSaveAs(RTMessage Msg) = [CM_FIRST + CM_SaveAs];
virtual void CMPrint(RTMessage Msg) = [CM_FIRST + CM_Print];
virtual void CMclose(RTMessage Msg) = [CM_FIRST + CM_Exit];

// Functions to handle 'geometry' menu
virtual void CMSpan(RTMessage Msg) = [CM_FIRST + CM_Span];
virtual void CMEndSpts(RTMessage Msg) = [CM_FIRST + CM_EndSpts];
virtual void CMMaterial(RTMessage Msg) = [CM_FIRST +
                                         CM_Materials];
// Function to handle material menu selection
void getMaterial(float, float);

// load cases menu
virtual void CMCases(RTMessage Msg) = [CM_FIRST + CM_Cases];

// load combination menu
virtual void CMComb(RTMessage Msg) = [CM_FIRST + CM_Comb];

// loading
virtual void CMUdl(RTMessage Msg) = [CM_FIRST + CM_Udl];
virtual void CMPart(RTMessage Msg) = [CM_FIRST + CM_Part];
virtual void CMConc(RTMessage Msg) = [CM_FIRST + CM_Conc];
virtual void CMLinear(RTMessage Msg) = [CM_FIRST + CM_Linear];
virtual void CMSettle(RTMessage Msg) = [CM_FIRST + CM_Settle];

// analysis menu
virtual void CMAalyze(RTMessage Msg) = [CM_FIRST + CM_Analyze];

// design menu
virtual void CMGivenSect(RTMessage Msg) = [CM_FIRST +
                                           CM_GivenSect];
virtual void CMMaxMmt(RTMessage Msg) = [CM_FIRST + CM_MaxMmt];
virtual void CMFinal(RTMessage Msg) = [CM_FIRST + CM_Final];

// view menu
virtual void CMNextView(RTMessage Msg) = [CM_FIRST + CM_NextView];
virtual void CMPrevView(RTMessage Msg) = [CM_FIRST + CM_PrevView];
virtual void CMCloseView(RTMessage Msg) = [CM_FIRST +
                                           CM_CloseView];

// help menu
virtual void CMAbout(RTMessage Msg) = [CM_FIRST + CM_About];

// Getting span data
void getSpanData(float[], float[],int);

// Starting shear design for the section
void shearDesign();

// Getting the data read from a data file
void getDatal();

```

```

        void getData2();
        void getData3();
        void getData4();

        // Transferring data to write data file
        void transferData1();
        void transferData2();
        void transferData3();
        void transferData4();

        // creating output file
        void outFile(char []);
};

_CLASSDEF(TFileDlg)

// class definition for the file dialog
class TFileDlg : public TFileDialog
{
public :
    TFileDlg (PTWindowsObject AParent, int ResourceId, LPSTR
                AFilePath)
        : TFileDialog(AParent, ResourceId, AFilePath){};
};

_CLASSDEF(TAboutDlg)

// class definition for the About dialog
class TAboutDlg : public TDialog
{
public :
    TAboutDlg (PTWindowsObject AParent, LPSTR ATitle) :
        TDialog (AParent, ATitle) {};
    void MyAboutOk(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
};

_CLASSDEF(TSpanDlg)

// class definition for the Span dialog
class TSpanDlg : public TDialog
{
private :
    int count;                // counter
    int NEL;                  // number of elements
    float span[12], I[12];    // span & inertia
    TEdit *spanEdit, *inertia; // edit controls for span & inertia
    TStatic *Static1;         // static control for span number
public :
    TSpanDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
        + WM_INITDIALOG];
    void SpanNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void SpanPrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
    void SpanDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void setSpan();
};

```

```

_CLASSDEF(TSptDlg)

// class definition for the Span dialog
class TSptDlg : public TDialog
{
private:
    int support[2];

public:
    TSptDlg (PTWindowsObject AParent, LPSTR ATitle) :
        TDialog (AParent, ATitle) {};
    virtual void RadioOne(RTMessage Msg) = [ID_FIRST + ID_RADIOONE];
    virtual void RadioTwo(RTMessage Msg) = [ID_FIRST + ID_RADIO TWO];
    virtual void RadioThree(RTMessage Msg) = [ID_FIRST +
        ID_RADIO THREE];
    virtual void RadioFour(RTMessage Msg) = [ID_FIRST + ID_RADIO FOUR];
    virtual void RadioFive(RTMessage Msg) = [ID_FIRST + ID_RADIO FIVE];
    virtual void RadioSix(RTMessage Msg) = [ID_FIRST + ID_RADIO SIX];

    virtual void sptCancel(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    virtual void sptDone(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST +
        WM_INITDIALOG];

    void setSpt();
};

_CLASSDEF(TMatDlg)

// class definition for the Material dialog
class TMatDlg : public TDialog
{
private :
    TEdit *Edit1, *Edit2; // edit boxes for fc & fy
    float fc,fy;

public :
    TMatDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
        + WM_INITDIALOG];

    void setMat();
    void matDone(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    void matCancel(RTMessage Msg) = [ID_FIRST + ID_Cancel];
};

_CLASSDEF(TCaseDlg)

// class definition for the load case dialog
class TCaseDlg : public TDialog
{
private:
    TEdit *Edit1; // edit box for load case
    TStatic *Static1; // static control for name
    char loadCase[10][70] ;
    int count, NEL;

public :
    TCaseDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
        + WM_INITDIALOG];

```



```

    void setCase();
    void caseDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void caseNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void casePrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
};

_CLASSDEF(TCombDlg)

// Combination dialog box class
class TCombDlg : public TDialog
{
private:
    int count, NEL;
    TEdit *Edit1, *Edit2;          // Edit boxes
    TStatic *Static1, *Static2;    // Static boxes
    TListBox *caseList;            // list box for load cases
    char loadCase[10][70], loadComb[10][70];
    float factor[10][10];

public:
    TCombDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setComb();
    void combDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void combNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void combPrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
    void addCase(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    void deleteCase(RTMessage Msg) = [ID_FIRST + ID_Delete];
    virtual void HandleListBoxMsg(RTMessage Msg)
        = [ID_FIRST + ID_List1];
};

_CLASSDEF(TUdlDlg)

// UDL dialog box class
class TUdlDlg : public TDialog
{
private:
    int count, NEL, udlType[12][5];
    TEdit *Edit1;                  // Edit box
    TListBox *caseList, *spanList; // list box for load cases & span
    char loadCase[10][70];
    float span[12], w[12][5];      // 5 udls per span are allowed

public:
    TUdlDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setUdl();
    void udlDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void udlNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void udlPrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
    virtual void HandleListBoxMsg(RTMessage Msg)
        = [ID_FIRST + ID_List2];
};

_CLASSDEF(TSettleDlg)

// UDL dialog box class

```

```

class TSettleDlg : public TDialog
{
private:
    int count, NEL;
    TEdit *Edit1;    // Edit box
    TStatic *Static1; // static control for the joint name
    float jtDisp[13];

public:
    TSettleDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setSettle();
    void settleDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void settleNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void settlePrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
};

_CLASSDEF(TPartDlg)

// Partial distributed load dialog box class
class TPartDlg : public TDialog
{
private:
    int count, NEL, partType[12][5];
    TEdit *Edit1, *Edit2, *Edit3;    // Edit box
    TListBox *caseList, *spanList;    // list box for load cases & span
    char loadCase[10][70];
    float span[12], partW[12][5], partA[12][5], partC[12][5];
                                // 5 partial loads per span are allowed

public:
    TPartDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setPart();
    void partDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void partNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void partPrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
    virtual void HandleListBoxMsg(RTMessage Msg)
        = [ID_FIRST + ID_List2];
};

_CLASSDEF(TConcDlg)

// Concentrated load dialog box class
class TConcDlg : public TDialog
{
private:
    int count, NEL, concType[12][10];
    TEdit *Edit1, *Edit2;    // Edit box
    TListBox *caseList, *spanList;    // list box for load cases & span
    char loadCase[10][70];
    float span[12], P[12][10], concA[12][10];
                                // 10 conc loads per span are allowed

public:
    TConcDlg (PTWindowsObject AParent, LPSTR ATitle);
    virtual void WMInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

```

```

        void setConc();
        void concDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
        void concNext(RTMessage Msg) = [ID_FIRST + ID_Next];
        void concPrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
        virtual void HandleListBoxMsg(RTMessage Msg)
            = [ID_FIRST + ID_List2];
};

_CLASSDEF(TLineDlg)

// Concentrated load dialog box class
class TLineDlg : public TDialog
{
private:
    int count, NEL, lineType[12][5];
    TEdit *Edit1, *Edit2, *Edit3, *Edit4;    // Edit box
    TListBox *caseList, *spanList;           // list box for load cases
                                              // & span
    char loadCase[10][70];
    float span[12], lineW1[12][5], lineA[12][5];
                                              // 5 varying loads per span are allowed
    float lineW2[12][5], lineC[12][5], b, d;

public:
    TLineDlg(PtWindowsObject AParent, LPSTR ATitle);
    virtual void WmInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setLine();
    void lineDone(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    void lineNext(RTMessage Msg) = [ID_FIRST + ID_Next];
    void linePrev(RTMessage Msg) = [ID_FIRST + ID_Prev];
    virtual void HandleListBoxMsg(RTMessage Msg)
        = [ID_FIRST + ID_List2];
};

_CLASSDEF(Ratio)

// Dialog class for steel ratio input from the user
class Ratio : public TDialog
{
private:
    TEdit *Edit1;
    float ratio, rhoMax, rhoMin;
public:
    Ratio(PtWindowsObject AParent, LPSTR ATitle);
    virtual void WmInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setData(float , float);
    void okPressed(RTMessage Msg) = [ID_FIRST + ID_Cancel];
};

_CLASSDEF (TSectionDlg)

// Section data dialog box
class TSectionDlg : public TDialog
{
private:
    TEdit *Edit1, *Edit2;
    TListBox *sectList;
    float trialB[7], trialD[7];
};

```

```

public :
    TSectionDlg(PtWindowsObject AParent, LPSTR ATitle);
    void sectOK(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    virtual void WmInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];

    void setData();
    void closeSect(RTMessage Msg) = [ID_FIRST + ID_Cancel];
    virtual void HandleListBoxMsg(RTMessage Msg)
        = [ID_FIRST + ID_List2];

};

// Class for the dialog object to warn the user about the doubly reinf.
section
_CLASSDEF (TWarnSectDlg)

class TWarnSectDlg : public TDialog
{
public :
    TWarnSectDlg :: TWarnSectDlg(PtWindowsObject AParent, LPSTR AName)
        : TDialog(AParent, AName){};

    void doublyOK(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    void closeDoubly(RTMessage Msg) = [ID_FIRST + ID_Cancel];
};

// Class for the stirrup data
_CLASSDEF (TStirrupDlg)

class TStirrupDlg : public TDialog
{
private :
    TEdit *Edit1;
public :
    TStirrupDlg(PtWindowsObject AParent, LPSTR ATitle);
    virtual void WmInitDialog(RTMessage Msg) = [WM_FIRST
                                                + WM_INITDIALOG];
    void stirrupOK(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    void stirrupClose(RTMessage Msg) = [ID_FIRST + ID_Cancel];
};

_CLASSDEF (TGiveSectDlg)

// Given Section data dialog box
class TGiveSectDlg : public TDialog
{
private :
    TEdit *Edit1, *Edit2;
    float b, d;
public :
    TGiveSectDlg(PtWindowsObject AParent, LPSTR AName);
    void sectOK(RTMessage Msg) = [ID_FIRST + ID_AboutOK];
    void closeSect(RTMessage Msg) = [ID_FIRST + ID_Cancel];
};

```

```

_CLASSDEF(readData)

// class for file reading & writing
class readData
{
    private :
        float span[12], I[12], fc, fy, factor[10][10], w[12][5],
            jtDisp[13];
        float partW[12][5], partA[12][5], partC[12][5], P[12][10],
            concA[12][10];
        float lineW1[12][5], lineW2[12][5], lineA[12][5],
            lineC[12][5];
        int NEL, combination, support[2], udlType[12][5],
            partType[12][5], concType[12][10], lineType[12][5];
        char loadCase[10][70], loadComb[10][70];

    public :
        void init ();
        void read (HWND);
        void write (HWND);

        // transfer from data file
        void out1();
        void out2();
        void out3();
        void out4();

        // write data to file
        void setData1();
        void setData2();
        void setData3();
        void setData4();
};

```

Appendix B

Table B.1 lists the major subroutines in Holzer's frame analysis program and the corresponding methods in the Analysis class of the analysis and design of reinforced concrete continuous beams application. Although to someone unfamiliar with the concepts of object-oriented programming it may appear that these functions are similar there are some fundamental differences between the two. These are outlined below.

1. Holzer's program is written in FORTRAN 77, a procedural programming language which is now over fifteen years old and has been replaced with FORTRAN 90.
2. The Actor version of Analysis is written using the Actor programming language. Since Actor is a pure object oriented programming language there is little similarity between the Actor code and the corresponding code in a procedural programming language such as FORTRAN 77. The C++ version of Analysis is written using AT & T's C++ version 2.0. Again there is little resemblance between C++ code and FORTRAN 77 code.
3. The entire analysis procedure in the Continuous Beam application (both Actor and C++ versions) has been implemented as a class. This means that the analysis procedure is totally self contained and is independent of other parts of the program. Both the data and the methods are defined within the class. It is possible to modify and even replace the entire Analysis class (with another one) without affecting any other parts of the program.
4. The functions in the class 'Analysis' do not have any arguments since all the necessary data for the analysis is defined as data members within the class. In Holzer's program it is necessary to pass arguments to each function.

Table B.1 Comparison of Subroutines in Holzer's Frame Analysis Program and Functions in the 'Analysis' class of the Continuous Beam Application

Holzer's program	'Analyze' Object	Purpose of function
Subroutine 'Codes'	Functions 'jCode' and 'mCode'	Generate Joint and Member code.
Subroutine 'MBAND'	Function 'halfBand'	Compute half band width.
Subroutine 'MACT'	Functions 'udlFixed', 'partFixed', 'concFixed' and 'linearFixed'	Compute fixed end actions due to member actions.
Subroutine 'ASSEMF'	Function 'generateQ'	Assemble local fixed end forces.
Subroutine 'ELEMS'	Function 'Assemble'	Compute global stiffness coefficients.
Subroutine 'ASSMS'	Function 'Assemble'	Assign system stiffness coefficients to global stiffness matrix.
Subroutine 'SOLVE'	Function 'Solution'	Call the functions to solve the stiffness equation.
Subroutine 'ELEMF'	Function 'eleF'	Compute local forces on elements.

5. Unlike Holzer's program there are no global variables or arrays and none are needed.

6. Since 'Analysis' is implemented as a class, it can easily be used in other programs such as, for example, a program to perform the analysis of plane frame or space frame. By using the inheritance feature it is possible to derive a new class based on 'Analysis' that provides whatever additional functionality is needed. It would not be necessary to make any changes to 'Analysis' itself. It is not possible to extend Holzer's program without making extensive modifications to the original code, a process that would be extremely difficult since it is written in FORTRAN and uses global variables.

7. The allocation of memory for objects of the 'Analysis' class is done dynamically. Memory is allocated as needed only during the analysis process. After the results of the analysis are saved, the entire analysis object can be removed from memory (along with the associated variables and arrays) thus making the additional memory available to the program for other tasks such as design.

8. The C++ version of 'Analysis' makes extensive use of pointers for assigning arrays, a feature that is not available in FORTRAN. The use of pointers makes it possible to create arrays dynamically and also results in faster execution especially when working with large matrices.

9. The subroutines in Holzer's program and in the 'Analysis' class are fairly standard and are available in numerous other frame analysis programs. All frame analysis programs provide subroutines for computing the half band width, computing fixed end forces, computing local stiffness matrices, assembling the global stiffness matrix, solving the resulting equations and computing local element forces.

10. Although the Continuous Beam Design application used a matrix analysis approach for performing analysis, there are many other analysis techniques that can be used for the analysis of continuous beams, such as for example, the three moment method. In fact, given the modular nature of the application, it would be a relatively easy task to replace the 'Analysis' class with one that uses a different analysis procedure.

Vita

Ajay B. Kulkarni was born on September 9, 1969, in Pune, India. He graduated from M. E. S. Boys' High School, Pune, in May 1984. In May 1987 he graduated from Cusrow Wadia Institute of Technology, Pune, with a Diploma in Civil Engineering. In May of 1990 he was awarded Bachelor of Civil Engineering by Pune University and received a Gold Medal for being first in the University. He joined the Construction Division of Tata Engineering and Locomotive Company, Limited (TELCO), Pune, as a Graduate Trainee Engineer. In August 1991 he enrolled in the Department of Civil Engineering at Virginia Polytechnic Institute and State University where he earned Master of Science in Civil Engineering in May 1993.

A handwritten signature in black ink, reading "Ajay B. Kulkarni", with a horizontal line underneath.