INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING, VOL. 35, 425–436 (1992)

OBJECT ORIENTED PROGRAMMING OF THE FINITE ELEMENT METHOD

R. I. MACKIE

Wolfson Bridge Research Unit, Department of Civil Engineering, The University, Dundee, DD1 4HN, U.K.

SUMMARY

The finite element method is by its nature very modular. Object oriented programming enables full advantage to be taken of this modularity. This makes for safer and easier programming, and extending or modifying object oriented programs is very straightforward. The paper describes an object oriented implementation of the finite element method, and illustrates the advantages of the approach.

INTRODUCTION

The finite element method is one of the most widely used numerical tools in engineering, particularly in structural analysis.[1,2] It is also applied to problems such as heat transfer and fluid mechanics, and to many other problems that are described by a set of differential equations. The reason for the widespread use of the finite element method is that it is both powerful and lends itself easily to modular programming. Almost all finite element problems include the following main steps:

(i) data input;
(ii) calculation of stiffness matrices and load vectors;
(iii) assembly of global stiffness and load vectors;
(iv) application of constraints;
(v) solution of equations;
(vi) results output.

The exact order and form in which these steps occur will vary from program to program (e.g. steps 2–5 may be done simultaneously as in the frontal solution method), but they form the basis for all finite element programs, be it dealing with a structural problem, fluid mechanics or whatever.

The finite element method was first used in the 1950's and its use increased rapidly in the 1960's. Up until a few years ago it was implemented almost exclusively on mainframe computers. The last ten years have seen the advent of the personal computer, and the phenomenal increase in its power, and decrease in cost, a development which is continuing unabated. One consequence of this is that finite element packages that were once confined to mainframes are becoming increasingly available on personal computers.

Coupled with the development of hardware has been the development of software and programming languages. The finite element method has traditionally been programmed in Fortran, and most standard texts on finite element programming include Fortran code.[2,3] Fortran's name is derived from 'formula translation', and Fortran was designed around that

0029–5981/92/150425–12$11.00
© 1992 by John Wiley & Sons, Ltd.

Received 10 July 1990
Revised 24 October 1991

concept. One of the major developments in programming languages is the introduction of 'object oriented programming' or OOP. OOP is designed around a very different concept, namely the intrinsic linking of data and subroutines. This paper will examine the benefits that can be attained by applying object oriented programming to finite element analysis.

OBJECT ORIENTED PROGRAMMING

Descriptions of the object oriented programming concept can be found in many computer journals[4–6] and language user guides. In the majority of them the descriptions are given in terms of graphics programs or general organizational problems, though there are some that deal with mathematical applications.[7,8] Here the concept will be described directly in terms of the finite element method. The hope is that this will enable engineering programmers to understand the concept and see its potential more easily. In this section a general description of OOP concepts will be given, then a more detailed study of an object oriented approach to finite element programming will be given in the next section. There are several object oriented programming languages ranging from specialist languages such as Smalltalk, to extensions of standard languages such as C + + . In this paper programming will be described in terms of Turbo Pascal. This is an implementation of Pascal that falls into the latter category. However, it is emphasized that the concepts described apply to any object oriented language.

In traditional programming languages, such as Fortan, data and subroutines are separate entities and are linked together only by the passing of data to a subroutine when it is called. Indeed the data structures in Fortran are also rather limited: integer, real, logical, character and arrays being the only types available in standard Fortran 77. Other languages allow more complex data structures such as records in Pascal. A record such as the following could be defined in Pascal:

```
elem_data = record
        no_nodes: integer;
        mat: integer;
        node_list: Anode_list;
end;
```
where
```
Anode_list = array[1. . .16] of integer;
```

In Fortran the three sets of data contained within *elem_data* would need to be stored separately and it would be up to the programmer to ensure that they always appeared together at the right time. With records the three sets of data are intrinsically linked and will always be together; this appears to be more logical as the sets of data are actually linked. However, the data and the procedures (Pascal's equivalent of subroutines) are linked only when the procedure is called. OOP goes one step further and intrinsically links procedures and data (see Figure 1); the procedures are commonly called *methods* in OOP terminology. So an object such as the following could be defined:

```
element = object
        data:elem_data;
        procedure get_stiff_mat(var nsize:integer; var kmat:Akmat);
end;
```
where
```
Akmat = array[1. . .1176] of double;
```

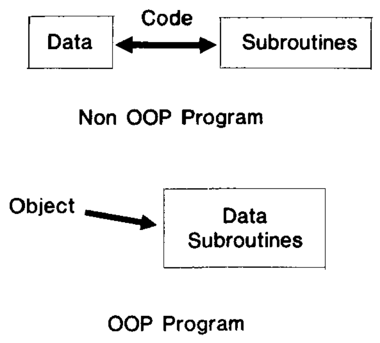

Figure 1. How data and subroutines are linked in OOP and non-OOP programs

(the number 1176 is the size needed to store a symmetric 48×48 matrix; if necessary the arrays *Anode_list* and *Akmat* could be made larger).

If *elem* was a specific instance of an *element* object a call to the method *get_stiff_mat* would look as follows:

```
elem.get_stiff_mat(n,kmat);
```

The part of the program making this call would then obtain the stiffness matrix for *elem*. Note that there was no need to explicitly pass the node list, material number, etc. to *get_stiff_mat* as the procedure and the data are already linked together.

Linking data and procedures is logical, and this should lead to clearer thinking and hence to better program design and fewer errors. However, the real power of OOP becomes evident only when two further features are considered: inheritance and virtual methods.

Inheritance and virtual methods

OOP allows descendants of objects to be defined and a descendent automatically inherits all the data and the methods belonging to its ancestor. However, some of the descendant's methods while being generally similar might be different in detail. The use of virtual methods allows this to be taken care of. Again consider a finite element. Certain things are true of all elements; for instance they all consist of a collection of nodes, are made of a certain material. So the data part of the definition of element given above applies to any sort of element. Any type of element object should inherit these data. In addition stiffness matrices can be obtained from any sort of element, but the precise way the stiffness matrix is calculated depends upon the type of element, e.g. beam, plane stress, plate, etc. So each type of element object would have a different method to calculate the stiffness matrix.

Now consider the assembly process in a finite element package. At some stage the assembler needs to obtain the stiffness matrix of each element, but the assembler is not in the least concerned about how the stiffness matrix was calculated, it only wants to know what it is. This is where virtual methods come in. Change the definition of element to the following:

```
element = object
  data : elem_data;
  procedure get_ stiff_ mat(var nsize:integer:var kmat:Akmat);
                                        virtual;
end;
```

This procedure *get_stiff_mat* could be a dummy, or abstract, method that did not actually do anything; all the definition is intended to do is to declare that stiffness matrices can be obtained from elements, which is all that the assembler requires. In itself this object is not particularly useful as at present it does not actually calculate stiffness matrices. This is done by defining descendants of *element*. For instance a plane stress element could be defined:

```
element_ps = object(element);
   procedure get_stiff_mat(var nsize:integer; var kmat:Akmat);
                                             virtual;
end;
```

element_ps is a descendant of *element* and so automatically inherits the data. It is also capable of giving a stiffness matrix, and the procedure *get_stiff_mat* would contain the code needed to calculate the stiffness matrix. Similarly beam and plate elements could be defined with their own procedures for calculating stiffness matrices.

Suppose that the program contains an array of pointers to *element* types:

```
elements = array[1. . .1000] of ^element;
```

(i.e. *elements[i]* points to the memory location of the *i*th object). Now the following definition is quite legal:

elements[i] = elem_ps (1)

where *elem_ps* is a pointer to type *element_ps*, i.e. it is legal to assign a descendant to its ancestor. Suppose this was done for all the entries in *elements* and *elements* was then given to the assembler. When the assembler came to try and obtain the stiffness matrix of each element the call to *get_stiff_mat* would automatically use the plane stress version of the procedure and so give the correct stiffness matrix. Similarly if elements had all been defined as beam elements the beam version of *get_stiff_mat* would have been used. This is because *get_stiff_mat* is a virtual method and the program only decides which particular version to use at runtime. Statements such as equation (1) let the program know that *element[i]* is a plane stress element and so when the program encounters the statement

```
elements[i]^.get_stiff_mat(nsize,kmat);
```

it knows which one to use, there is no need to use 'if' statements: (*elements[i]* points to the memory location of the *i*th element, *elements[i]^*, while *get_stiff_mat* tells the computer to use the *get_stiff_mat* method for element *i*.)

The above is a very brief description intended to give a quick introduction to the concepts of object oriented programming. A more detailed account will now be given, though owing to constraints of space and the need to preserve clarity it is of course not possible to include everything.

FINITE ELEMENT PROGRAMMING

Basic element

The most important step in designing an object oriented program is to decide what each object does. In terms of finite elements an element is used primarily by the equation assembler/solver (henceforth this will be referred to simply as the 'solver'). The solver requires the following information from an element:

(i) the variables associated with that element;
(ii) the stiffness matrix.

Also every element has a number of nodes and is made of some material or other, so these will be included as data. Now define an element object as follows:

```
element = object
        no_nodes:integer;
        mat:integer;
        node_list:Anode_list;
        constructor init(qnodes,qmat:integer; qlist:Anode_list);
        destructor done;
        get_var_list(var nsize:integer; var var_list:Alist); virtual;
        get_stiff_mat(var nsize:integer; var kmat:Akmat); virtual;
end;
```

where

```
Anode_list = array[1. . .16] of integer;
     Alist = array[1. . .48] of integer;
    Akmat = array[1. . .1176] of double;
     Aload = array[1. . .48] of double;
```

(The array sizes have been chosen so that they are large enough to be able to handle any problems the program is likely to encounter.) Here *no_nodes*, *mat* and *node_list* are the object's data, and *init*, *done*, *get_var_list* and *get_stiff_mat* are the object's methods: *init* is a special method called a constructor; this initializes the object. So if element 47 had 4 nodes, a node list (1,5,89,145) and material number 2 then *init(47,4,qlist)* (where qlist = [1,5,89,145]) sets *element's* data to the appropriate values: *init* also tells the program which virtual methods to use; this will be explained further when *element_ps* is described. Here *done* is also a special method and is called when the object is no longer needed; however, for present purposes it is of little importance. The methods *get_var_list* and *get_stiff_mat* would be abstract methods, i.e. they tell the program what it can except an object of type element, and any of its descendants, to do. The solver needs only to know how to get a stiffness matrix from an element; it is not interested in the details of how that stiffness matrix is derived. Given the above definition of element the solver routine could be written as it now knows how to obtain the stiffness matrices and variable lists as and when necessary: i.e. *element* is a base object whose prime purpose is to define inheritable characteristics for its descendants, rather than to be used directly.

2-d element

Clearly for the program to be any use it will need to have some 'real' elements in it. Suppose that the program is going to be used initially to solve plane stress/strain problems, but that at some future date it may be extended to other two dimensional problems. There are certain features that are common to most two dimensional elements, therefore a general object will be defined:

```
element_2d = object(element)
        get_elem_xy(var elem_xy:Aelem_xy); virtual;
        calc_xy_derivs(s,t:double; var det:double; var xydv:Aelem_xy;
                                var error:integer); virtual;
        calc_det2(s,t:double; var det:double; var xydv:Alem_xy;
                                var error:integer); virtual;
        shape2d(s,t:double; var shape:Ashape); virtual;
        shapedv2d(s,t:double; var shapedv:Aelem_xy); virtual;
end;
```

where

```
    Aelem_xy = array[1. . .16,1. . .2] of double;
      Ashape = array[1. . .16] of double;
```

get_elem_xy returns the x–y co-ordinates of the element: *calc_xy_derivs* returns the x–y derivatives of the shape functions and the determinant of the Jacobian at the parent co-ordinate point (s,t), *calc_det* returns the determinant of the Jacobian without calculating the x–y derivatives, while *shape2d* and *shapedv2d* return the shape function values and the derivatives of the shape functions respectively. The code *object* (*element*) in the declaration of *element_2d* indicates that it is a descendant of *element*, and therefore automatically inherits *element*'s data and methods.

Two of the methods will be looked at in more detail in order to make the ideas a little more concrete. First of all consider *calc_det*, which could be coded as follows:

```
procedure element_2d.calc_det;
var
  elem_xy, xydv, shapedv:Aelem_xy;
  jacob:array[1. . 2,1. . .2] of double;
  i,j,inod:integer;
  q1,q1:double;
  begin
  get_elem_xy(elem_xy);
  shapedv2d(s,t,shapedv);
  for i:= 1 to 2 do
        begin
        q1:= 0;
        q2:= 0;
        for inod: = 1 to no_nodes do
           begin
           q1:= q1 + shapedv[inod,i]*elem_xy[inod,1];
           q2:= q2 + shapedv[inod,i]*elem_xy[inod,2];
           end;
        jacob[i,1]:= q1;
        jacob[i, 2]:= q2;
           end-;
        det:= Jacob[1,1]*jacob[2,2]-jacob[1,2]*jacob[2,1];
        end;
```

There are a number of points to note:

(i) *element_2d.calc_det* is Turbo Pascal syntax for telling the compiler that the ensuing code is the method *calc_det* belonging to object *element_2d*. The argument list does not need to be included again as it has already been defined in the object declaration;
(ii) the method uses the objects methods *get_elem_xy* and *shapedv2d*;
(iii) the method knows about the variable *no_nodes* because the variable *no_nodes* is part of the object *element*, and *element–2d* inherits all *element*'s data and methods (though virtual methods can be overridden if necessary);
(iv) the argument list is relatively small; this is because much of the data used by methods is either part of the object (e.g. *no_nodes*), or is obtained using the object's methods (e.g. *get_elem_xy*). An advantage of this is that subroutine calls are less dependent on how data are stored in the program, and so changes to say how the nodal co-ordinates were stored would not necessitate changes to all parts of the program where they were needed, but only to the one method *get_elem_xy*. This makes program changes easier to deal with, and less likely to introduce bugs.

Now consider the shape function method *shape 2d*. Suppose that initially it was assumed that the program was going to have 3-, 4-, 6- and 8-node elements. Then *shape 2d* would have the following form:

```
procedure element_2d.shape2d;
begin
if no_nodes = 3 then
  begin
  .
  end
else if no_nodes = 4 then
  begin
  .
  end
else if no_nodes = 6 then
  begin
  .
  end
else if no_nodes = 8 then
  begin
  .
  end;
end;
```

The actual code that gives the shape functions for each case is very similar to typical Fortran code and so has not been included: *shapedv2d* would be similar in structure.

All *element_2d*'s methods have been declared virtual. It is probably safe not to declare *calc_det* as virtual as it will work for all forseeable elements: however, declaring a method as virtual makes the object more flexible and allows the previously unforeseen circumstance that may arise to be dealt with easily without changing existing methods. For instance the methods *shape2d* and *shapedv2d* assume that an element has either 3, 4, 6 or 8 nodes. Because these methods have been declared to be virtual this is not an irrevocable decision, as will be demonstrated later when the introduction of a 9-node element is considered.

Plane stress element

The object element_*2d* still is not fully functional as it cannot calculate stiffness matrices, etc. This facility has not been included upto this point because none of the other methods of the element depended upon whether the element was a plane stress, plate bending, heat flow element or whatever. Now a specific element, namely one for plane stress, will be developed. Define *element_ps* as follows:

```
element_ps= object(element_2d)
        get_var_list(var nsize:integer; var var_list:Alist); virtual;
        get_stiff_mat(var nsize:integer; var kmat:Akmat); virtual;
end;
```

Much of the functionality of the element has already been described in *element_2d;* this does not need to be recoded as *element_ps* inherits it from *element_2d.* For instance the method *get_stiff_mat* will use the method *calc_xy_derivs* (and therefore *shapedv* as well). The methods listed above are those that were included in the declaration of the base object *element*; the difference is that in *element* they were abstract methods, indicating what an element could do, but not how it did it. In *element_ps* code to calculate the stiffness matrices, etc. for a plane stress element will be included. Again this will be similar to typical Fortran code except that where subroutines would be called in Fortran, methods (e.g. *calc_xy_derivs*) will be used, and the argument lists will be shorter. Note that because *element_ps* is a descendent of *element–2d* it is also a descendent of *element*, and therefore automatically inherits *element's* data.

There are two variables associated with each node, the x and y displacements u and v. If these are listed u_1,v_1, u_2,v_2, . . . , etc., *get_var_list* could be written as follows:

```
procedure element_ps.get_var_list;
var
   inod,iv:integer;
begin
nsize:= 0;
for inod:= 1 to no_nodes do
   begin
   nsize:= nsize + 1;
   var_list[nsize]:= 2*node_list[inod]-1;
   nsize:= nsize + 1;
   var_list[nsize]:= var_list[nsize – 1] + 1;
   end;
end;
```

element_ps is now fully functional. Suppose that element 47 is a 4-node plane stress element made of material 2 and having node list (1,5,89,145). Suppose that

```
Pelement_ps= ^element_ps
```

i.e. *Pelement_ps* is defined as pointer type and points to objects of type *element_ps*. Then the following code would make *elements[47]* be an object of type *element_ps:*

```
qlist[1]:= 1;   qlist[2]:= 5;   qlist[3]:= 89;   qlist[4]:= 145;
elements[47]:= new(Pelement_ps, init(4,2,qlist));
```

The above code does three things:

(i) the word *new* is Turbo Pascal syntax for telling the computer to create a new object;
(ii) it sets the data of element 47 to the appropriate values;
(iii) it tells the computer that element 47 is a plane stress element. This means that when, for instance, the solver wants the element 47's stiffness matrix the correct method is used, i.e. the program knows which virtual methods to use.

The above description has concentrated on the stiffness matrix, but code to determine load vectors for elements could be included in a similar manner. In a complete program there would be many more details such as data input to be included; these details have not been included as they would obscure the main purpose of the paper, i.e. to describe the concept of object oriented programming. Also in the above description fixed arrays were used for *node_list*, *kmat*, etc. It is more efficient to use dynamically sized arrays.

EXTENDING THE OBJECT

So far code has been described for a specific implementation, namely plane stress elements. However, one of the main advantages of the object oriented approach is the relative ease with which existing code can be extended. Some examples of this will now be given.

9-node element

Suppose it is desired to include a 9-node plane stress element. The object *element_2d*'s methods for shape functions (*shape2d* and *shapedv2d*) dealt only with 3-, 4-, 6- and 8-node elements. If the programmer had access to the source code for *element_2d* he could change the source code, but there are a number of reasons for preferring not to do this: (i) any changes to existing code, however simple, may introduce new bugs into the program; (ii) the programmer may possess the code only in machine code that can be linked into his own programs but cannot be altered. Can a 9-node element be added without changing the original source code? The answer is yes. Suppose that the programmer has machine code for *element_2d* and *element_ps*. A new element could now be defined:

```
element_ps9 = object(element_ps)
        shape2d(s,t:double; var shape:Ashape); virtual;
        shapedv2d(s,t:double; var shapedv:Aelem_xy); virtual;
end;
```

The methods shape2d and *shapedv2d* would be written with the code for 9-node shape functions. Then if element 47 is a 9-node element it would be defined as

```
element[47]:= new(Pelement_ps,init(9,2,qlist));
```

The key points to note are that existing code has not been changed, and the existing code will use the new code when necessary. For example, if an element is of type *element_ps* the method *calc_xy_derivs* will use the code for *shapedv2d* that was written for *element_2d*, but if the element is of type *element_ps9* then *calc_xy_derivs* will use the new routines even though the writer of the code for *calc_xy_derivs* knew nothing about the new routines. That is object oriented programming allows new code to use existing code, and allows old code to use new code, and does this without the programmer needing access to the source code.

An alternative, and better, way of achieving the same result is to define an object as follows:

```
element_psA = object(element_ps)
        shape2d(s,t:double; var shape:Ashape); virtual;
        shapedv2d(s,t:double; var shapedv:Aelem_xy); virtual;
end;
```

The method *shape2d* would be as follows:

```
procedure element_psA.shape2d;
begin
if no_nodes < = 8 then
   element_2d.shape2d(s,t,shape)
else
   {code for 9 node shape functions}
end;
```

shapedv2d would be written in a similar way. The object *element_psA* encapsulates 3-, 4-, 6-, 8- and 9-node elements, whereas *element_ps9* includes only the 9-node element. The advantage of *element_psA* is that if some further development of the object is made, then if the new object is defined as a descendant of *element psA* it will work for 3-, 4-, 6-, 8- and 9-node elements.

Plate bending element

Plate bending elements would be defined in much the same way as plane stress elements were originally defined, i.e.

```
element_pb = object(element_2d)
        get_var_list(var nsize:integer; var var_list:Alist); virtual;
        get_stiff_mat(var nsize:integer; var kmat:Akmat); virtual;
end;
```

The appropriate code would be written in the same way as for *element_ps: element_pb* inherits *element_2d*'s methods, but not those of *element_ps.*

Dynamic problems

Suppose that the original code was written for static problems, but it is later decided to include dynamic problems as well. Code would now be required to calculate mass and damping matrices. How could this be incorporated into the object structure? One alternative would be to define a new object *element_ps_dyn*:

```
element_ps_dyn = object(element_ps)
        procedure get_mass_mat(var nsize:integer; var mass:Akmat); virtual;
        procedure get_damp_mat(var nsize:integer; var damp:Akmat); virtual;
end;
```

and write the appropriate code for the new methods. Some solution routine, say, *dyn_solver*, would be written that accepted elements of type *element_ps_dyn.*

```
dyn_solver(elements:Aelements_ps_dyn);
```
where
```
Aelements_ps_dyn = array[1. . .1000] of ^element_ps_dyn;
```

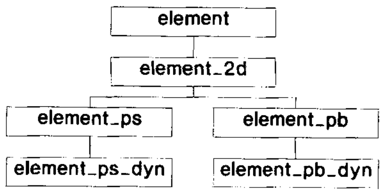

Figure 2. Object tree for element objects

This would work fine if the program was going to be used for dynamic plane stress problems only; but suppose that it was desired to solve dynamic plate bending problems as well. The obvious answer would be to define an object *element_pb_dyn:*

```
element_pb_dyn= object(element_pb)
        procedure get_mass_mat(var nsize:integer; var mass:Akmat); virtual;
        procedure get_damp_mat(var nsize:integer; var damp:Akmat); virtual;
end;
```

This raises a problem, however. The dynamic solution routine *dyn_solver* accepts objects of type *element_ps_dyn* and its descendants, but *element_pb_dyn* is not a descendant of *element_ps_dyn* (see Figure 2), so *dyn_solver* cannot be used. In object oriented programming a descendent can be assigned to its ancestor, but not vice versa. Suppose *elem* is of type *element*, and *elemdyn* is of type *element_pb_dyn* then the following is legal:

```
elem:= elemdyn;
```

but the following is not:

```
elemdyn:= elem;
```

This problem could be overcome if the original definition of *element* had included abstract methods to calculate mass and damping matrices, i.e.

```
element= object
        no_nodes:integer;
        mat:integer;
        node_list:Anode_list;
        constructor init(qnodes, qmat:integer; qlist:Anode_list);
        destructor done;
        procedure get_var_list(var nsize:integer; var var_list:Alist); virtual;
        procedure get_stiff_mat(var nsize:integer; var kmat:Akmat); virtual;
        procedure get_mass_mat(var nsize:integer; var mass:Akmat); virtual;
        procedure get_damp_mat(var nsize:integer; var damp:Akmat); virtual;
end;
```

The fact that the additional routines were never going to be used in a static analysis program does not matter; their inclusion makes the object more flexible in case they might be used in the future. With the new definition the dynamic solver just needs to accept objects of type *element*, and *element_ps_dyn* and *element_pb_dyn* can be defined as above with code for calculating the mass and damping matrices.

The 9-node element example illustrates that it is possible to extend objects in ways that the original programmer may not have envisaged, and the above example illustrates that it still pays to think ahead. OOP does not remove the need for thinking ahead; indeed it increases the rewards for doing so, but it is impossible to foresee all possible needs or uses of code. Object oriented programs can cope with these situations.

Declaring methods to be virtual does involve a small computational cost in that whenever a virtual method is to be used the program has to decide which particular version of the method to use. However, this cost is very small and must be set against the fact that the program will not include as many 'if blocks'.

DISCUSSION AND CONCLUSIONS

The finite element method is by its very nature a modular numerical tool; an object oriented approach allows full advantage to be made of this modularity. Some details of an object oriented implementation of the finite element method have been presented. The purpose was to illustrate how the concepts of object oriented programming can be applied to the finite element method; it is not the only-possible object oriented implementation. An object oriented approach reduces the scope for bugs by encouraging clearer thinking about program design, and allowing programs to be substantially altered without the need to change existing code. However, object oriented programming does not obviate the need for forward thinking, nor does it make it impossible to introduce bugs. Billions of pounds has already been invested in Fortran finite element programs and it is not suggested that this work be scrapped! Indeed well tried and tested Fortran code is less likely to contain bugs than new object oriented code. However, there is still scope for the advantages of object oriented programming to be utilized. If new systems are developed then serious consideration should be given to using objects. There is increasing interest in standardization and quality assurance in finite elements,[9] the possible relevance of object oriented programming to this should be considered. On a smaller scale, research workers who want to use novel elements could derive considerable benefit. An object oriented program would mean that new elements and adaptations of existing elements could be tried with a much reduced chance of introducing bugs. So the researcher would be free to concentrate on the new element and its coding without wondering if he has introduced new bugs in the existing code. Object oriented programming would also allow very flexible tool boxes to be developed.

REFERENCES

1. O.C. Zienkiewicz, *The Finite Element Method*, 3rd edn, McGraw-Hill, New York, 1977.
2. K.J. Bathe, *Finite Element Procedures in Engineering Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
3. E. Hinton and D.R.J. Owen, *Finite Element Programming*, Academic Press, New York, 1977.
4. E. Gibson, 'Objects—Born and bred', *Byte*, **15**, 245–254 (1990).
5. N. Hampshire, 'Object of the exercise', *Personal Comp. World*, **13**, (8) 174–180, (1990).
6. J. Micallef, 'Encapsulation, reusability and extensibility on object oriented programming languages', *J. Object Oriented Programming*, **1**, (1), 12–38 (1988).
7. T.A. Budd, 'Generalized arithmetic in C + + ', *J. Object Oriented Programming*, **3**, (6), 11–23 (1991).
8. R.I. Mackie, 'Object oriented programming and numerical methods', *Microcomp. Civil Eng.*, **6**, (2), 123–128, (1991).
9. J. Maguire, 'Structural dynamics, finite element analysis and ISO 9001' in M. Petyt *et al.* (eds.), Elsevier *4th Int. Conf. on Structural Dynamics: Recent Advances*, 1991, pp. 332–339.