Microcomputers in Civil Engineering 6 (1991) 123–128

REVIEW ARTICLE

Object Oriented Programming and Numerical Methods

R. I. Mackie

Wolfson Bridge Research Unit, Department of Civil Engineering, The University, Dundee, DD1 4HN, UK

Abstract: *The paper examines the ways in which the object oriented approach to programming can be applied to engineering problems. The principles of object oriented programming are described in terms of the finite element method, and an object oriented implementation of an equation solver for an MS-DOS system is described in detail. These two examples are used to illustrate the added flexibility that can be gained by using the object oriented approach when writing numerical method programs.*

INTRODUCTION

One of the fields of software development that is currently very active is that of object oriented programming (OOP). The key feature of OOP is that data and procedures are intrinsically linked. This will be described in detail a little later, but the main advantages of this are that it allows the production of more flexible and less bug prone code. OOP can be implemented either via languages specifically designed around the OOP concept (e.g. 'Smalltalk') or by extensions to more traditional languages (e.g. C++ and Pascal). Most of the current applications of OOP appear to be graphics and database programs. This paper will look specifically at OOP in the context of numerical methods, i.e. in the context of a field that is perhaps more familiar to engineers.

The last 10 years have seen an enormous increase in the power of the personal computer (PC), a trend which is continuing. This has meant that computers have changed from being remote devices that are used only by specialists to being something that almost anyone can have ready access to. This has been coupled with great improvements in the quality and user-friendliness of available software. Following some way behind this trend is a similar development in finite element (FE) packages. Originally FE packages were of necessity implemented solely on mainframe computers; they also tended to be difficult and tedious to use. Now there is an increasing number of them available on PCs, and data entry and results analysis (pre- and post-processors) are becoming much easier to use.

FE programs, and indeed most numerical method software, are written predominantly in Fortran. In the early days of FE software this was the best way of doing it. This paper will describe how an OOP approach can be applied to engineering software based around numerical methods such as the FE method. For the sake of definiteness all programming will be described in terms of Turbo Pascal (version 6·0) on an MS-DOS system, but it is emphasised that the concepts illustrated apply to OOP languages in general.

One particular example that will be studied is the implementation of an equation solver, such as is at the heart of every FE program, on an MS-DOS based PC. Two of the important features of MS-DOS are: (i) MS-DOS works in memory segments of up to 64 kb in size; (ii) MS-DOS cannot directly access more than 640 kb of memory. This means that MS-DOS programs are better at handling arrays of size less than 64 kb than they are at handling larger ones. Some compilers need to be told that a program contains large arrays, while others such as Turbo Pascal do not allow any one data item to occupy more than 64 kb, and two or more 'arrays' would have to be used to hold a single large array. Whatever the case, using large arrays involves an additional overhead in MS-DOS. FE programs often use large amounts of data, breaking the 640 kb limit. Many computers now have significantly more than 640 kb of memory, and MS-DOS can access this memory using expanded memory managers (EMS), but again there will be some loss in speed. These three situations will be

Microcomputers in Civil Engineering 0885-9507/91/$03.50

referred to as 'memory models', and will be called small, large and EMS:

1. Small—all arrays less than 64 kb.
2. Large—arrays greater than 64 kb.
3. EMS—use expanded memory.

A program that used EMS all the time could be written. Such a program could solve both large and small problems, but when solving small problems would be unnecessarily slow. On the other hand, a program that used a small memory model would be severely limited in the size of problem it could deal with. The ideal solution is a program that decides which memory model to use only when it knows the size of problem that it is dealing with at the time. This paper will describe how such a program can be written using OOP. OOP is not the only way of doing it, but it is the most elegant way.

The power of PCs, specifically the development of the 386 and 486 chips, has now outstripped the power of MS-DOS. However, the sheer weight of investment of time and money in MS-DOS software means that it is likely to remain the most common PC operating system for some time yet. Therefore, the work that is presented here is of direct use. However, the main purpose of the paper is to illustrate how an OOP approach can be of benefit to engineering programming, and the examples described should be viewed primarily in this light.

OBJECT ORIENTED PROGRAMMING

Descriptions of the OOP concept can be found in many computer journals[3,4,6] and language user guides. In the majority of them the descriptions are given in terms of graphics programs or general organisational problems, though there are some that deal with mathematical applications.[2] Here the concept will be described directly in terms of the FE method. The hope is that this will enable engineering programmers to understand the concept and see its potential more easily.

In traditional programming languages, such as Fortran, 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: integers, reals, logical, characters and arrays being the only types available in standard Fortran 77. Other languages allow more complex data structures such as records in Pascal. So a record such as the following could be defined in Pascal:

```
elem_data = record
  material : integer;
  no_nodes : integer;
  node_list: Anode_list;
end;
```

where

```
Anode_list = array[1..8] 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 only linked when the procedure is called. OOP goes one step further and intrinsically links procedures and data—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 k:Akmat);
end;
```

where

```
Akmat = array[1..126] of single;
```

(The number 126 is the size needed to store a symmetric 16×16 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 explicit need to pass the node list, material number, etc., to *get_stiff_mat* as the procedure and the data are automatically 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 only becomes evident when two further features are considered: inheritance and virtual methods.

Inheritance and virtual methods

OOP allows descendants of objects to be defined, and a descendant 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 FE. Certain things are true of all elements, for instance, they all consist of a collection of nodes and are made of a certain material. So the data part of the definition of an element given above applies to any sort of element. Any

type of element object should inherit this data. In addition, stiffness matrices can be obtained from any sort of element, but the precise way the stiffness matrix is calculated depends on the type of element, e.g. beam, plane stress or plate. So each type of element object would have a different method to calculate the stiffness matrix.

Now consider the assembly process in a FE package (this may, of course, be an integral part of the equation solver, e.g. if the frontal solution method[5] is used). 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 k:Akmat);
                          virtual;
end;
```

This procedure *get_stiff_mat* could be a dummy 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 k: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;
```

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 eqn (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, and there is no need to use 'if' statements.

In a complete FE package the element object would contain many more methods. Only one has been shown above, so that the object concepts are not obscured by too much detail.

In order to give further insight into the object oriented approach, an object oriented equation solver will be described in some detail. This example is sufficiently complex as to be worthwhile, but simple enough that the main points are not obscured by detail.

AN OBJECT ORIENTED EQUATION SOLVER FOR MS-DOS SYSTEMS

Finite element equations are commonly solved by using either banded solvers, frontal solvers or active column solvers. The key feature of them all is that they seek to take advantage of the sparsity of FE equations (i.e. the high percentage of zeroes that occur). The active column solver will be considered here.

The details of the method can be found in many standard FE textbooks[1] and only the most important features will be mentioned here. The essence of the method is to decompose the stiffness matrix, **K**, as follows:

$$\mathbf{K} = \mathbf{L}^t \mathbf{D} \mathbf{L}$$

where **L** is a lower triangular matrix with ones on the diagonal, and **D** is a diagonal matrix. The sparsity of **K** is taken advantage of by using a skyline storage scheme. The $\mathbf{L}^t\mathbf{D}\mathbf{L}$ decomposition can be carried out, using the following algorithm:

for $j = 2, \ldots, n$

$$g_{m_j,j} = k_{m_j,j}$$

$$g_{ij} = k_{ij} - \sum_{r=m_m}^{i-1} l_{ri} g_{rj} \quad i = m_j + 1, \ldots, j-1$$

where m_j is the row number of the first nonzero element in column j, and

$$m_m = \max\{m_i, m_j\}$$

$$l_{ij} = \frac{g_{ij}}{d_{ii}} i = m_j, \ldots, j-1$$

$$d_{jj} = k_{jj} - \sum_{r=m_j}^{j-1} l_{rj} g_{rj}$$

The l_{ij} are the entries in $\mathbf{L}^t$. The quantities g_{ij} are intermediate quantities only, and the calculated values of l_{ij} and d_{jj} immediately replace the values k_{ij} and k_{jj}, so only the one matrix **K** needs to be used. At the start of the process, **K** is the structural stiffness matrix, and at the end the diagonal contains **D** and the off-diagonal entries contain $\mathbf{L}^t$.

The elements will typically be stored by columns in a single array, say **A**, and an integer array will contain indices of where one column begins and another ends. The $\mathbf{L}^t\mathbf{DL}$ decomposition preserves the skyline shape. In the present context the key feature is the need to access the array **A**. Traditionally, one would simply define an array of sufficient size to handle the largest problem the program is designed to cope with, and the matrix is accessed directly. As mentioned earlier, on an MS-DOS system there are reasons for wanting a more flexible system. Such a system can be designed, using an object oriented approach if a different view of a matrix is taken. Consider the following typical operations that might be needed to be carried out on **A**:

```
q:=A[i];
A[i]:=q;
A[i]:=A[i]+q;
```

Instead of using an array, an object could be defined as follows:

```
gen_array = object
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
end;
```

The function *get_val* returns the value at location *i*, *set_val* sets the value at location *i* to *q*, and *add_to* adds *q* to the value at location *i*. So, if instead of **A** being a straightforward array, **A** is defined to be a *gen_array*, then the three statements above would be replaced by:

```
q:=A.get_val(i);
A.set_val(i,q);
A.add_to(i,q);
```

So anywhere in a solution algorithm the traditional array statements could be replaced by the appropriate object statement. The active column solver could be written in terms of objects; *gen_array* would then be extended as follows:

```
gen_array = object
  no_eqns : integer;
  index : Iarray;
  constructor init(n:integer; idx:Iarray);
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
  procedure solve_eqn(var x:Rarray); virtual;
end;
```

no_eqns is the number of equations and *index* the array used to indicate where the various columns are stored; *init* is used to initialise the object; *solve_eqn* is a method containing the active column solution algorithm where array operations are written in object terms; and *x* is an array which on entry contains the load vector, and on exit contains the solution.

The above definition of *gen_array* says that a value at location *i* can be accessed, etc., but does not actually say how it is done. What now needs to be done is to define *get_val*, *set_val* and *add_to* for each of the three memory models. So the following objects are defined:

```
gen_array_small = object(gen_array)
  data : big_array
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
end;
gen_array_large = object(gen_array)
  data : array[1..5] of big_array_ptr;
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
end;
gen_array_ems = object(gen_array)
  data : ems_stream;
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
end;
```

big_array is an array of real numbers whose size is less than 64 kb; *big_array_ptr* is a pointer to a *big_array*. This allows *big_arrays* to be dynamically allocated, so *gen_array_large* allocates as many *big_arrays* as it needs, providing there is enough memory. *ems_stream* is a means of accessing EMS memory.

gen_array_small uses a single array of size less than 64 kb, which is the sort that MS-DOS likes best. For this case the methods would be:

```
gen_array.small.get_val;
var
  ii : integer;
begin
ii:=i;
get_val:=data[ii];
end;
gen_array_small.set_val;
var
```

```
  ii : integer;
begin
ii: = i;
data[ii]: = q;
end;
gen_array_small.add_to;
var
  ii : integer;
begin
ii: = i;
data[ii]: = data[ii] + q;
end;
```

The line *ii*: = *i* is needed in Turbo Pascal because array indices must be of integer type. For *gen_array_small* the index *i* will always be small enough to be stored as an integer, but for the other two memory models this is not the case, so *i* is declared as a *longint* (4 byte integer) type.

Corresponding routines would be written for *gen_array_large* and *gen_array_ems*. With *gen_array_large*, *get_val* would decide which of *big_arrays* to use, and which entry of that array to access. For *gen_array_ems* the form of the methods would depend on the EMS access method used. For instance, data could be like a direct access file and *get_val* and *set_val* would be like read and write statements.

Note that *solve_eqn* does not have to be rewritten, this is because it has been written in terms of the object methods. So if **A** is of type *gen_array_small*, it would use the *gen_array_small* versions of *get_val*, etc. If **A** was of type *gen_array_ems* the *solve_eqn* would automatically know to use the EMS versions.

To implement the above equation solver one would make the following declaration:

```
A : Pgen_array;
```

Pgen_array is a pointer to *gen_array*. Suppose *nsize* is the problem size. Once *nsize* has been determined, the following lines could be included:

```
if nsize < = small_limit then
  A: = new(Pgen_array_small,init(n,index))
else if nsize < = large_limit then
  A: = new(Pgen_array_large,init(n,index))
else
  A: = new(Pgen_array_ems,init(n,index));
```

Pgen_array_small, etc., are pointers to *gen_array_small*, etc. The above lines tell the program what sort of *gen_array* **A** actually is, and hence which versions of *get_val*, etc., to use. *set_val* and *add_to* could then be used to put in the appropriate stiffness matrix terms. Once this has been done, solving the equations would merely require the line:

```
A^.solve_eqn(R);
```

The important point to note is that the only decision lines that needed to be put in by the programmer are the ones above which allocate *a*; the object programming language automatically takes care of the rest. This makes the programming easier and reduces the scope for introducing bugs. The only difference between the three memory models is the means of accessing data. The solution algorithm is exactly the same and, using an OOP approach, only needs to be written once. The programmer needs to define the appropriate means of data access for the various memory models, and does not have to work out where all the places in the program are that this might make a difference, i.e. 'if' statements do not need to be added to take care of the various possibilities.

COMPUTATIONAL COST OF USING VIRTUAL METHODS

One objection that might be raised is that the method of implementing *gen_array_small* is unnecessarily involved when all the program needs to do is to access a single array, and there will be an associated cost in computation time. It is true that there is some overhead involved in implementing virtual methods, but this has to be balanced against the increase in flexibility. Moreover, steps can be taken to reduce this overhead. For instance, there are stages in the algorithm where a contiguous sequence of numbers need to be accessed. Instead of accessing each one individually, methods could be written to access them all at once. So the object definition would now become:

```
gen_array = object
  no_eqns : integer;
  index : Iarray;
  constructor init(n:integer; idx:Iarray);
  function get_val(i:longint):single; virtual;
  procedure set_val(i:longint; q:single); virtual;
  procedure add_to(i:longint; q:single); virtual;
  procedure solve_eqn(var x:Rarray); virtual;
  procedure get_values(i:longint; n:integer; var rec);
                                    virtual;
  procedure set_values(i:longint; n:integer; var rec);
                                    virtual;
end;
```

The methods *get_values* and *set_values* access *n* contiguous values beginning at location *i*, *get_values* retrieves these into *rec*, and *set_values* writes the values in *rec* into the array. Different versions of each of these routines would be written for each of the memory models. If it was felt that the cost of using the above access methods was too high for the small memory model, then a special version of *solve_eqn* could be written for *gen_array_small*, which used direct array access instead of the methods *get_val*, etc. The definition of *gen_array_small* would then be:

```
gen_array_small = object(gen_array)
  data: big_array;
  constructor init(n: integer; idx: Iarray);
  function get_val(i: longint): single; virtual;
  procedure set_val(i: longint; q: single); virtual;
  procedure add_to(i: longint; q: single); virtual;
  procedure solve_eqn(var x: Rarray); virtual;
  procedure get_values(i: longint; n: integer; var rec);
                                virtual;
  procedure set_values(i: longint; n: integer; var rec);
                                virtual;
end;
```

It is important to note that giving *gen_array_small* its own *solve_eqn* method is the only change that is needed. There would be no changes elsewhere in the program.

In order to give some idea of the computational times involved, calculations were carried out on a thin plate subjected to uniform plane stress.

Symmetry was used so that a quarter of the plate was analysed. The plate was split up into a uniform mesh of 225 four-node elements, with the nodes numbered by rows. Calculations were carried out on a 16 MHz 80386 machine using various memory models. The results are shown in Fig. 1. Small 2 refers to *gen_array_small*, using methods to access data, and small 1 refers to *gen_array_small*, using direct array access. Using the latter method produces a saving of about 2·4 s. Small 2 is about 0·4 s quicker than the large model, and the EMS model is about 6·2 s slower than the large model. For the above problem, memory model small 1 clearly gives the best results, but if a larger problem were to be solved this model would not be able to handle it. While the EMS model is significantly slower than the large model, it can handle much larger problems.

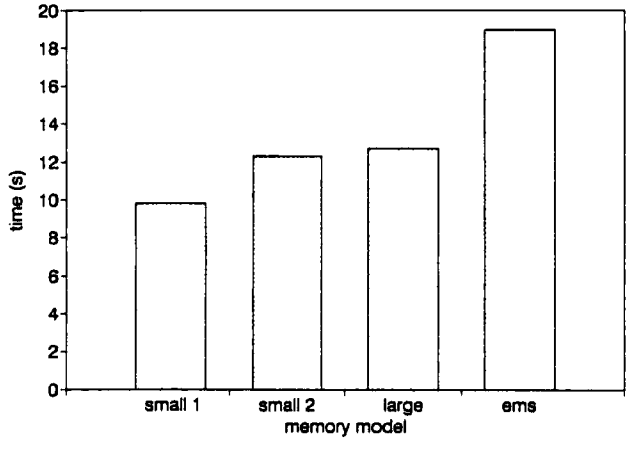

Fig. 1. Comparison of calculation times.

SUMMARY AND CONCLUSIONS

The paper has described how the object oriented approach to programming can be used in programs centred around numerical methods. OOP allows flexible programs to be written more easily, and makes for less bug prone programs. The main advantage comes when a part of a program is doing essentially the same thing, but where the details may vary from application to application. One example is finite elements where all elements give stiffness matrices and load vectors, but the way these are calculated varies from case to case. The example of an equation solver was presented in some detail. The same algorithm was used throughout, but different data access methods were used, depending on problem size.

There is some overhead involved in using objects, but against this is the fact that it allows a program to make the best use of the facilities available on the computer. The examples described in the paper are useful in their own right, but it is hoped that they give engineers an appreciation for how an object oriented approach to programming can offer advantages.

REFERENCES

1. Bathe, K. J., (ed.) *Finite Element Procedures in Engineering Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
2. Budd, T. A., Generalized arithmetic in C++. *J. Object Oriented Programming*, 3(6) (1991) 11–23.
3. Gibson, E., Objects—born and bred. *Byte*, (Oct.) **15** (1990) 245–54.
4. Hampshire, N., Object of the exercise. *Personal Computer World*, 13(8) (1990) 174–80.
5. Irons, B. M., A frontal solution program for finite element analysis. *Int. J. Num. Meth. Engng*, **2** (1970) 5–32.
6. Micallef, J., Encapsulation, reusability and extensibility on object oriented programming languages. *J. Object Oriented Programming*, **1** (1) (1988) 12–38.