Received March 1998
Revised November 1998
Accepted November
1998

Why you should consider object-oriented programming techniques for finite element methods

J.T. Cross, I. Masters and R.W. Lewis
Department of Mechanical Engineering, University of Wales Swansea, Swansea, UK

Keywords *Finite elements, Heat transfer, Programming, Solidification*

Abstract *Object-oriented programming, as an alternative to traditional, procedural programming methods for finite element analysis, is growing rapidly in importance as algorithms and programs become more complex. This paper reviews some of the literature and seeks to explain some of the concepts of object-oriented thinking most useful to the finite element programmer, using as an example a C++ implementation of a heat transfer and solidification program.*

1. Introduction

Even relatively simple finite element programs can be quite complex pieces of software, and the highest code writing standards are required if one developer's work is to be understood and re-used successfully by another. This is exacerbated by the incorporation of more sophisticated algorithms and by the desire for increased usability and robustness, which, perversely, makes it even more important that software developers can share and re-use code.

Most finite element programs are written in procedural languages (like Fortran or C) in which the finite element algorithm is broken down into procedures (functions, subroutines etc.) that manipulate data (numbers, character strings, vectors, matrices etc.). Every time a procedure is used, the user (the code that calls it) must provide it with the correct data, which generally requires that the user knows not just what the procedure does, but how it does it. As procedures become more complex and their interactions become more deeply nested, this knowledge becomes increasingly hard to acquire.

The purpose of this paper is not to give a detailed explanation of how to implement a finite element algorithm in an object-oriented language, but to explain why you should consider doing so. It will show that taking an object-oriented approach to finite element code development has significant advantages and, in particular, allows rapid program development through code sharing and re-use. The main disadvantage is that poor design early in the development process can be hard to correct later.

This paper is based on work first presented at the 10th International Conference for Numerical Methods in Thermal Problems, Swansea, July 1997 (Cross *et al.*, 1997; Masters *et al.*, 1997).

© MCB University Press, 0961-5539

The next section will report on some of the uses that have been made of object-oriented techniques in numerical methods programming. Section 3 will briefly explain some of the basic concepts of object-oriented programming, while Section 4 will explain some aspects in more detail, and will show how they relate to finite element programming by using, as an example, a heat transfer and solidification program written in C++.

2. Uses of object-oriented techniques in numerical methods

One of the earliest ways in which object-oriented methods have been used in numerical methods programming is in the graphical generation and visualisation of data. The windows, buttons and scroll bars found on any graphical user interface form natural "objects", and geometric primitives such as tetrahedra, cubes and cylinders on can be thought of in the same way. An increasing number of object-oriented packages have been written specifically for finite element data, an example of which is FEView (Zheng *et al.*, 1995).

Probably the most common use of object-oriented techniques is in the development of the ubiquitous "integrated packages" – along the authors' own office corridor there are at least three[1,2] (Marchant *et al.*, 1996). In most cases these packages are graphical interfaces designed to simplify the assembly of data files and to monitor the running of otherwise standard sequential finite element codes. Whilst not appealing to the object-oriented purist, these systems often use object-oriented ideas (sometimes without the authors even realising it!) and they are a very practical means of making finite element codes more usable and less error prone. One advantage of using an object-oriented language for the analysis code as well as the graphics code is that it opens up the prospect of much more integrated, "integrated packages". Several researchers (Shah *et al.*, 1994; Miller *et al.*, 1995; Ju and Hosain, 1996; Bettig and Han, 1996) are currently exploring this exciting new area.

A need common to many areas of numerical computing is to manipulate vectors and matrices. Scholtz (1992) explains very clearly how the four basic operators (add/subtract, multiply/divide) can be re-programmed for such data types in C++, while Zeglinski *et al.* (1994) show a number of further examples. This is usually termed operator overloading and has the advantage of making numerical code much more compact and easier to follow. It is a sufficiently useful concept that it is now a part of Fortran90 (Smith, 1995) and C++ (Stroustrup, 1991) object libraries for matrix manipulation are available commercially[3].

A further level of abstraction can be achieved by considering symbolic processing packages such as Mathematica[4] and Matlab[5]. Although these are used for serious research, they often run slowly in comparison to other methods and tend to be used primarily for non-computationally demanding applications. However, Viklund *et al.* (1992) and Fritzson *et al.* (1994) produced their own language, ObjectMath, which is compatible with Mathematica for symbolic manipulation but which can produce C++ code for fine tuning. Zimmermann and Eyheramendy (1996) and Eyheramendy and Zimmermann

(1996a; 1996b) have developed a high level system with a user interface into which a user can enter a set of governing equations. The program then creates a new object containing the corresponding numerical model, which can be slotted into an existing analysis framework.

The design of objects is a core activity in object-oriented finite element programming, but one in which an accepted wisdom has yet to emerge. Several strategies have been proposed to describe the inter-relationships between objects (Mackie, 1992; Zimmermann *et al.*, 1992; Kong and Chen, 1995), with each author having a different approach. Nevertheless, object-oriented finite element analyses have been used in a number of application areas including: stress analysis (Dubois-Pelerin *et al.*, 1992; Dubois-Pelerin and Zimmermann, 1993), hypersonic shock waves (Budge and Peery, 1993), structural dynamics (Pidaparti and Hudli, 1993), shell structures (Ohtsubo *et al.*, 1993), non-linear plastic strain (Mentrey and Zimmermann, 1993), electromagnetics (Silva *et al.*, 1994) and contact problems (Feng, 1995). The application used as an example later in this paper is a heat and fluid flow program, of which there are few reported in the literature. Peskin and Hardin (1996) have considered fluid flow in the context of electroplating, and include a model of the electrochemical reaction. The authors' own contributions examine solidification problems (Masters, 1997) and have also developed a version for parallel architectures (Masters *et al.*, 1997).

3. Object-oriented programming

3.1 Overview

An object-oriented program is a collection of intelligent, interacting objects. An object can be likened to a data structure with its own built-in procedures (often called methods). An object is an instance of a class and a class can inherit features of another class from which it is derived. This means that, once some basic classes have been designed, new classes can be generated very quickly. Objects can pass messages to each other, but data can only be manipulated by the object containing that data. This is known as encapsulation. The object sending the message is usually called the client. Almost anything can be an object and designing flexible and powerful classes/objects is the key to producing successful and re-usable code.

3.2 Objects and classes

The best way to think of an object is as a "service provider that is alive, responsible and intelligent" (Cline and Lomov, 1995). It provides a service to its clients (the objects that use it) by meeting clearly defined objectives known as a specification. It is alive because it is able to construct and initialise itself, live a full and productive life, and take care of its own demise. An object is responsible because it monitors the integrity of its own data and will not accede to any request that would violate that integrity. It is intelligent because it is capable of carrying out the user's instructions without the user needing to know how those instructions are carried out.

At the risk of trivialising the profound difference between the object-oriented and procedural programming paradigms, Figure 1 shows an attempt to explain it by means of an analogy.

3.3 Encapsulation

An object is an instance of a class and is defined by its attributes (data) and its methods (functions). All of the attributes and, frequently, some of the methods are private, that is, they cannot be seen or altered by any other object. Communication with clients takes place only through public methods, which must provide all of the services a client might need. This is called encapsulation and sounds very restrictive but, in fact, most objects need to provide

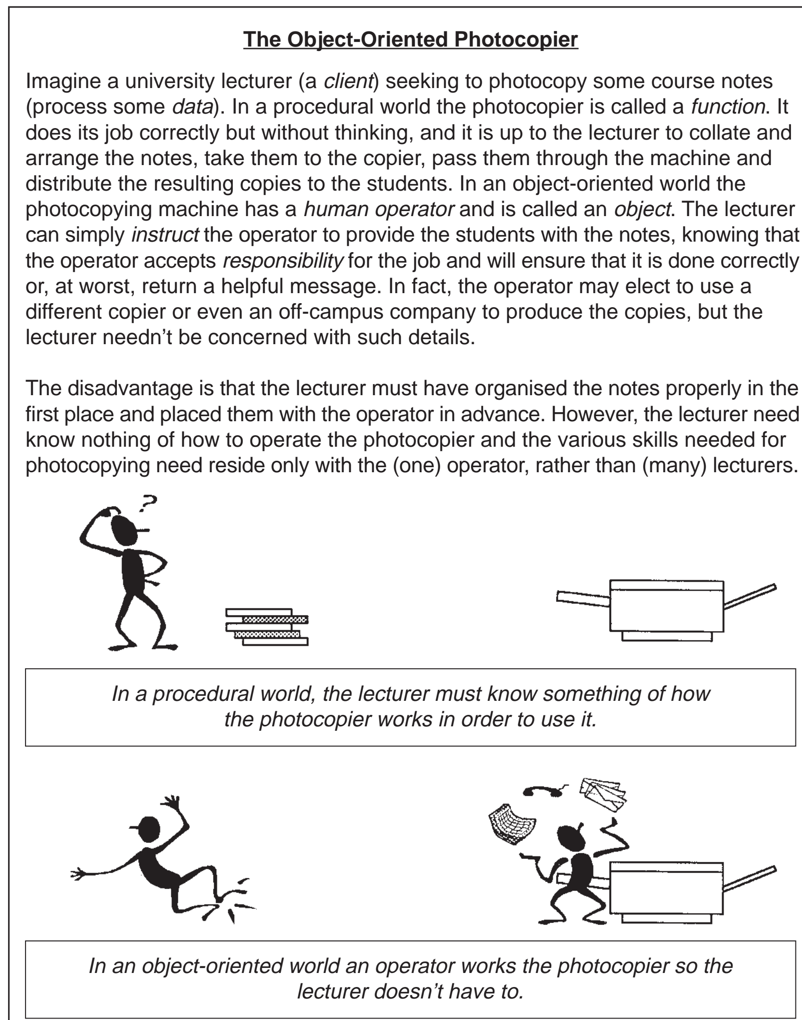

Figure 1.
An analogy to explain the difference between procedural and object-oriented philosophies

surprisingly few external services and, in a finite element program, they are usually fairly easy to define. The very significant advantage is that the person designing the class can create and test objects in isolation, safe in the knowledge that it is impossible to impose unplanned demands on an object or give it incorrect data. This is shown graphically in Figure 2.

A class is the specification of an object, which is what makes it possible to separate the purpose of a function (what it does) with its actual implementation (how it does it). For example, a mesh object will probably contain a method to read in a mesh from a file. All the client needs to know is that there is a method called `readmesh()` that will perform that task. The client does not need to know how it does it, how many nodes and elements there are, what order they are in, or even how the program stores the data after it has been read. It is even quite possible to re-code parts of a class at a later date (to change the way the data is stored perhaps) without the client code needing to be re-compiled.

3.4 Inheritance

When one class is derived from another one, it inherits all of the attributes and methods of that class. The derived class or subclass can also have additional attributes and methods, and it can redefine those that are not appropriate. In a procedural language a function copied and edited by the programmer also "inherits" most of the properties of the original. However, once altered it is generally no longer fit for its original purpose, resulting in two similar but independent functions. Inheritance is a formal and controllable way of developing classes stage-by-stage, while still being able to use any of the intermediate versions.

3.5 Polymorphism

Three things identify an object-oriented method (function); its name, the object it is attached to and the data passed to it. It is, therefore, possible for functions

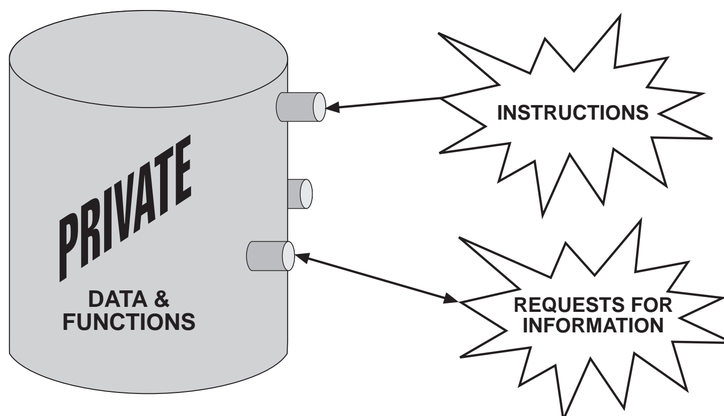

Figure 2. An object is protected by only being able to provide services through its public functions

with the same name to exist as long as they are attached to different objects or accept different arguments. This enables the same client instruction (function call) to invoke different methods depending on the related object. This is called polymorphism. It can be applied to operators as well as function calls, in which case it is known as operator overloading.

4. Implementation in C++ of a heat transfer finite element program

The purpose of this section is to explain how a finite element program can be written using object-oriented techniques and to give an indication of the advantages that might accrue from taking such an approach. Since the main task is to design the classes/objects, that is the aspect we will concentrate on.

The example used is a finite element program written in C++ (by far the most widely used object-oriented language) to solve heat transfer with solidification using the enthalpy approach. A detailed appreciation of the algorithms is given elsewhere (Lewis *et al.*, 1996) but is not necessary for the purposes of this paper. A limited number of code fragments are given, but the interested reader is directed to the authors' web site[6] where more detailed information can be found.

4.1 Classes for finite element analysis

The types of object classes used for finite element modelling fall into two categories. Mathematical classes such as matrices, vectors and tensors provide a flexible and high level way of manipulating finite element equations. Operations such as multiply and divide can be performed using operator overloading and member functions (methods) can perform inversion, transposition and other common matrix operations. These have quite a wide application outside finite element methods and have received more attention as a result. Geometrical classes are used to describe the problem domain and its boundary conditions. Examples include element, node, material, boundary and mesh objects, each of which will have member functions to manipulate data and perform the analysis. These are largely unique to finite element analysis and are the ones we shall focus on in the following descriptions.

Time class. This class is a relatively simple one and is a good example to choose for showing a code fragment. It contains data concerning start and finish times, timestep sizes and the current time, as well as the methods required for updating the current time, checking for convergence, adjusting timestep sizes and so on. In fact it is the only class concerned with the passage of time. The class definition is as follows:

```
Line01: class Time{
Line02:   protected:
Line03:     double start_time;     // time parameters
Line04:     double runtime;
Line05:     double timestep;
```

```
Line06:   int  step_number;
Line07:   double stop_time;
Line08:   double alpha;
Line09:   int  max_iterations;//non linear parameters
Line10:   int  max_restarts;
Line11:   double tolerance;
Line12:   int  niter_increase_timestep;
Line13:   int  niter_decrease_timestep;
Line14:   double max_timestep;
Line15:   double timestep_change_factor;
Line16:   int  iteration;
Line17:   int  restart;
Line18:   double last_norm;
Line19:   double this_norm;
Line20:   int  do_output;
Line21:   int  is_mesh_output;
Line22:   double last_output_time;
Line23:   double output_time_interval;
Line24:
Line25: public:
Line26:   Time();        // constructors
Line27:   Time(char* dataFileName);
Line28:   int  stop();
Line29:   int  output();
Line30:   int  mesh_output();
Line31:   int  mesh_written();
Line32:   int  increment(Mesh_ha& globalmesh);
Line33:   double getstepsize();
Line34:   int  write_step_info(const int n,FILE *fp);
Line35:   int  is_step_output();
Line36:   void  show();
Line37: };
```

The code fragment is shown largely for the sake of completeness, and a line-by-line dissection would not be appropriate. However, it can be seen that Lines 3-

23 contain declarations of parameters that are protected, i.e. they can only be "seen" by methods within the same object. Lines 26-36 contain declarations of public methods that provide the services to other objects.

Element class. Attributes for this class include nodal data, material properties and shape functions. Anything that is dependent on the element type is included in this class and, as a result, is used throughout the program – a fact reflected in the relatively large number of public methods. These include:

- `read_element_data( )` and `write_element_data( )` used for transferring information to and from file.
- `calc_shape_functions( )` used to calculate and store the appropriate shape functions and their derivatives. The calculation of the Jacobian matrix is also required, achieved using the function `calc_Jacobian( )`.
- `calc_element_matrix( )` forms the stiffness matrix for the element.
- `calcBC( )` applies flux boundary conditions to one or more element faces.
- `calc_fixedBC( )` applies fixed temperature boundary conditions to one or more element nodes.

The Element class is a good way to demonstrate the advantages of abstract data types and inheritance. A finite element code may well offer a choice of element types and it makes good sense to design for this facility from the start. One way to do this would be to design an abstract Element class containing declarations for the methods and attributes likely to be needed by an element object. The Element class would never itself be used but would provide features for derived classes such as Element 3 and Element 4 (three- and four-node elements) to inherit. This inheritance relationship is shown graphically in Figure 3. A client wishing to use an element object knows that all of the methods declared in the Element class must be supplied by any class derived from it. Thus, client code using only those services offered by the Element class is automatically compatible with any element type derived from it, even those

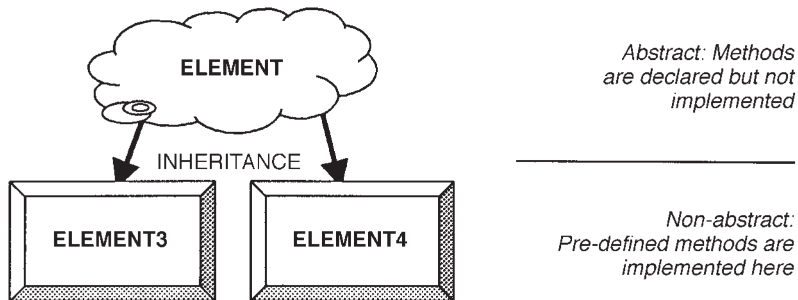

Figure 3.
How object classes inherit from abstract base classes

not yet written. This means that adding a new element type to an object-oriented finite element program is a self-contained process involving only the declaration of the derived element class and the writing of the code that implements the required functions. The author need know nothing of the client code and the client code need not even be re-compiled. This is somewhat of a contrast to the effort involved in adding a new element type to a typical, procedural finite element program.

Node class. Elements contain nodes and these are contained within a separate class. The data includes the node number, coordinates and temperature, while methods mostly deal with file reading and writing and accessing nodal data.

Material class. An instance of this class would contain all the information relating to one material and would provide methods to allow other objects to access that information. These methods can be used to demonstrate some of the advantages of encapsulation.

The solidification model uses an enthalpy method and the element calculations require the enthalpy of the material for a given temperature. In fact, enthalpy is a non-linear material characteristic for the phase change material and a linear material for the others. However, the element object (the client) is not interested in whether the material is non-linear or not and certainly does not want to have to provide code to determine the nature of the element's material before it can call the correct function. Because the information is thoroughly encapsulated the client can call the member function `get_enthalpy(temperature)` safe in the knowledge that a suitable technique will automatically have been used to calculate the enthalpy. It could be said that the Material class has assumed responsibility for ensuring that only correct material information is released to its clients.

Boundary class. Contains the data required for boundary conditions and all the methods necessary to apply them. This includes methods for applying non-uniform and non-linear boundary conditions, which are, again, transparent to the user.

GaussPoint class. Contains gauss data and methods for providing an element with appropriate shape functions and derivatives.

Mesh class. This class contains data and methods that concern the mesh as a global entity, rather than at the element or node level. Thus, it holds the number of elements and nodes and maintains the global lists of elements and nodes. Its methods perform the following key tasks:

- file input/output (using node and element methods);
- organising the construction (and destruction) of the appropriate number of element and node objects;
- assembling the global stiffness matrix; and
- solving the global stiffness matrix.

4.2 The main ( ) routine

Every complete computer program needs a procedure that is activated first. In FORTRAN it consists of the code that is not part of a subroutine or function, in C++ it is a method or function called `main()`. The `main()` method in our example finite element program appears as follows.

```
Line01: #include "classdef.h"     //object definitions
Line02:
Line03: main(int argc, char **argv)
Line04: {
Line05:  Mesh           globalmesh;   //create objects
Line06:  SparseMatrix   global_matrix;
Line07:  Time           time;
Line08:  Material       materials;
Line09:  Gauss          gpdata;
Line10:  File           file;
Line11:  Vector globalRHS();
Line12:
Line13:  file.openfiles();         //preprocessing
Line14:  globalmesh.readmeshdata(file);
Line15:  globalmesh.assign_materials(materials);
Line16:  globalmesh.read_boundary_data(file);
Line17:  global_matrix.allocate_memory(globalmesh);
Line18:  globalmesh.preprocessor(materials,gpdata);
Line19:
Line20:  do                  //main loop
Line21:   {
Line22:   globalmesh.form_global_matrix
Line23: (time,materials,gpdata,global_matrix,globalRHS);
Line24:   globalmesh.solver(global_matrix,globalRHS);
Line25:   time.increment(globalmesh);
Line26:   globalmesh.write_results(time,file);
Line27:   }
Line28:  while( !time.stop(globalmesh) );
Line29: }
```

On Line 1 the header files containing the class definitions are included, while Line 3 marks the beginning of the `main()` function. Lines 5-11 contain the declarations in which objects are instantiated from classes which, in plain English, means that `globalmesh` is an object of class `Mesh` and so on. Lines 12-18 contain all the commands required to set up the solution process, while Lines 22-26 contain the statements that form the main iterative loop of the process. Line 29 marks the end of the `main()` function.

The syntax is relatively straightforward and can be deduced even by the reader for whom C++ is unfamiliar. Taking Line 24 as an example, `globalmesh` is the object, `solver` is the function, and `global_matrix` and `globalRHS` are the two objects being passed to the function.

The diagram in Figure 4 represents the main structure of the program in a graphical form and shows the relationships between the objects.

5. Conclusions

This paper has attempted to show that there is much to be gained by taking an object-oriented approach to writing finite element software. As usual, however, there are disadvantages too.

The hardest aspect of object-oriented programming is designing classes, which, unfortunately, is done right at the start of the development process, before any coding takes place. Designing effective classes requires a working knowledge of the chosen language and a thorough understanding of the finite element method. Designing really good classes requires, in addition, a thorough understanding of the object-oriented concept, and little flair.

In addition, to continue the photocopier analogy, if it was just one lecturer who needed notes copied, and copied only once at that, it probably wouldn't be worthwhile employing and training an operator for the machine. Likewise, developing a set of C++ classes for a relatively small, one-off computer program is not a very effective use of resources.

If the patterns of development of numerical analysis programs developed using sequential and object-oriented methods could be plotted, they might look something like the graph shown in Figure 5. For a sequential approach, research effort yields immediate benefits, but sustained or large group effort often yields diminishing returns. For an object-oriented approach, few gains are apparent initially but later, rapid and sustainable growth can be achieved as researchers build fruitfully upon what their colleagues have done before.

In summary, if a planned finite element program will have any of the following features:

- consist of more than, say, 5,000 lines of code;
- have complex logical structures;
- involve more than one developer;
- are likely to require modification at any point in the future;

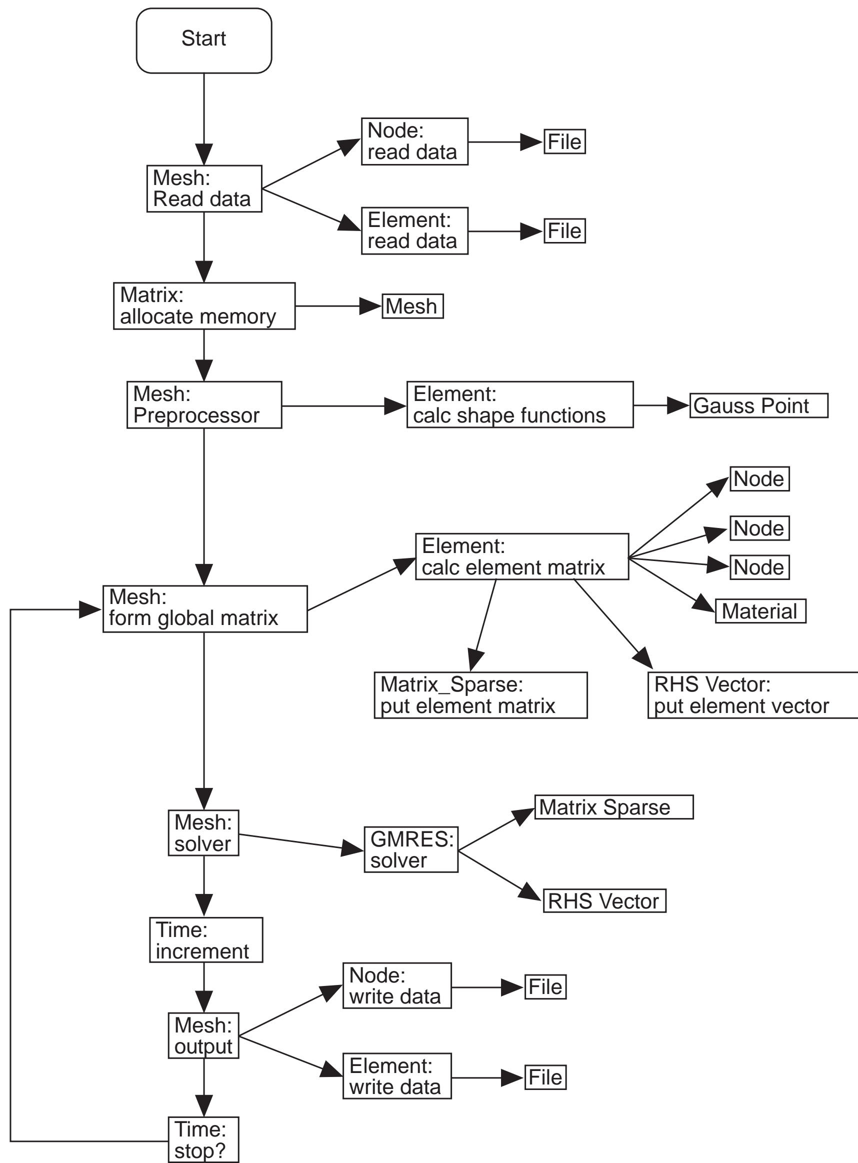

Figure 4.
Diagram showing the relationships between objects in a finite element heat transfer program

- contain any re-usable parts;
- need to be integrated with graphical user interfaces;
- need to be especially robust or error tolerant;

then an object-oriented approach is worthy of serious consideration.

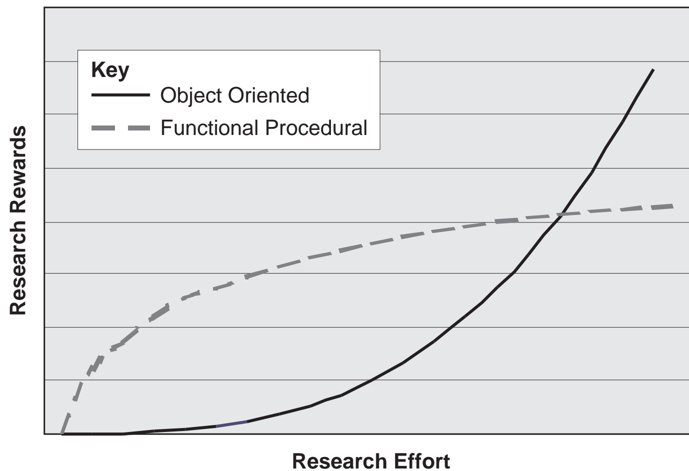

Figure 5.
A schematic graph of Research Effort versus Research rewards for object-oriented and functional procedural approaches to numerical code development.

Notes

1. Merlin Castings Analysis Software at http://www.swan.ac.uk/civeng/research/casting/merlin/index.html
2. FIDO, part of the ADOPT group at http://www.swan.ac.uk/civeng/Research/adopt/fido/index.html
3. The Math.h++ library from http://www.roguewave.com
4. Wolfram Research at http://www.wri.com
5. http://www.mathworks.com
6. http://www.swan.ac.uk/mecheng/sofer/

References

Bettig, B.P. and Han, R.P.S. (1996), "An object-oriented framework for interactive numerical analysis in a graphical user interface environment", *International Journal for Numerical Methods in Engineering*, Vol. 39, pp. 2945-71.

Budge, K.G. and Peery, J.S. (1993), "RHALE – A MMALE shock physics code written in C++", *International Journal of Impact Engineering*, Vol. 14, pp. 107-20.

Cline, M.P. and Lomov, G.A. (1995), *C++ FAQs: Frequently Asked Questions*, Addison Wesley, pp. 41-3.

Cross, J.T., Masters, I., Sukirman, Y. and Lewis, R.W. (1997), "Object oriented programming techniques for finite element methods in heat transfer", *10th International Conference for Numerical Methods in Thermal Problems*, Swansea, UK, July 21-25, Pineridge Press, ISBN 0-906674-89-1.

Dubois-Pelerin, Y. and Zimmermann, T. (1993), "Object-oriented finite element programming 3. An efficient implementation in C++", *Computer Methods in Applied Mechanics and Engineering*, Vol. 108, pp. 165-83.

Dubois-Pelerin, Y., Zimmermann, T. and Bomme, P. (1992), "Object-oriented finite element programming 2. A prototype program in Smalltalk", *Computer Methods in Applied Mechanics and Engineering*, Vol. 98, pp. 361-97.

Eyheramendy, D. and Zimmermann, T. (1996a), "Object-oriented finite element programming – an interactive environment for symbolic derivations, application to an initial boundary problem", *Advances in Engineering Software*, Vol. 27, pp. 3-10.

Eyheramendy, D. and Zimmermann, T. (1996b), "Object-oriented finite elements 2. A symbolic environment for automatic programming", *Computer Methods in Applied Mechanics and Engineering*, Vol. 132, pp. 277-304.

Feng, Z.Q. (1995), "2D or 3D frictional contact algorithms and applications in a large deformation context", *Communications in Numerical Methods in Engineering*, Vol. 11, pp. 409-16.

Fritzson, D., Fritzson, P., Viklund, L. and Herber, J. (1994), "Object-oriented mathematical modelling – applied to machine elements", *Computers and Structures*, Vol. 51, pp. 241-53.

Ju, J.N. and Hosain, M.U. (1996), "Finite element graphic objects in C++", *Journal of Computing in Civil Engineering*, Vol. 10, pp. 258-60.

Kong, X.A. and Chen, D.P. (1995), "An object-oriented design of FEM programs", *Computers and Structures*, Vol. 57, pp. 157-66.

Lewis, R.W., Morgan, K., Thomas, H.R. and Seetheramu, K.N. (1996), *The Finite Element Method in Heat Transfer Analysis*, Wiley, New York, NY.

Mackie, R.I. (1992), "Object-oriented programming of the finite element method", *International Journal for Numerical Methods in Engineering*, Vol. 35, pp. 425-36.

Marchant, M.J., Weatherill, N.P., Turner-Smith, E., Zheng, Y. and Sotirakos, M. (1996), "A parallel simulation user environment for computational engineering", *Proc. 5th Int. Conf. on Num. Grid Generation in Comp. Field Simulation*, Mississippi, April.

Masters, I. (1997), "Parallel Heat Transfer", PhD Thesis, University of Wales, Swansea.

Masters, I., Cross, J.T. and Lewis, R.W. (1997), "A review of object oriented programming techniques in finite element methods", *10th International Conference for Numerical Methods in Thermal Problems*, Swansea, UK, July 21-25, Pineridge Press, ISBN 0-906674-89-1.

Masters, I., Usmani, A.S., Cross, J.T. and Lewis, R.W. (1997), "Finite element analysis of solidification using object-oriented and parallel techniques", *Int. J. Num. Meth. Eng.*, Vol. 40 No. 2891-2909, ISSN 0029-5981.

Mentrey, P. and Zimmermann, T. (1993), "Object-oriented non-linear finite element analysis – application to J2 plasticity", *Computers and Structures*, Vol. 49, pp. 767-77.

Miller, G.R., Banergee, S. and Sribalaskandara, K. (1995), "A framework for interactive computational analysis in geomechanics", *Computers and Geotechnics*, Vol. 17, pp. 17-37.

Ohtsubo, H., Kawamura, Y. and Kubota, A. (1993), "Development of the object-oriented finite element modelling system – modify", *Engineering with Computers*, Vol. 9, pp. 187-97.

Peskin, A.P. and Hardin, G.R. (1996), "An object-oriented approach to general-purpose fluid-dynamics software", *Computers and Chemical Engineering*, Vol. 20 No. 8, pp. 1043-58.

Pidaparti, R.V.M. and Hudli, A.V. (1993), "Dynamic analysis of structures using object-oriented techniques", *Computers and Structures*, Vol. 49, pp. 149-56.

Scholz, S.P. (1992), "Elements of an object-oriented FEM++ program in C++", *Computers and Structures*, Vol. 43, pp. 517-29.

Shah, K.P., Meguid, S.A. and Zougas, A. (1994), "Development of software for integrated dynamic analysis of multibody systems", *Computer Aided Design*, Vol. 26, pp. 109-18.

Silva, E.J., Mesquita, P., Saldanha, R.R. and Palmeira, P.F.M. (1994), "An object-oriented finite element program for electromagnetic field computation", *IEEE Transactions on Magnetics*, Vol. 30, pp. 3618-21.

Smith, I.M. (1995), *Programming in Fortran 90*, Wiley, New York, NY.

Stroustrup, B. (1991), *The C++ Programming Language*, Addison-Wesley, Reading, MA.

Viklund, L., Herber, J. and Fritzson, P. (1992), "The implementation of ObjectMath – a high level programming environment for scientific computing", *Lecture Notes in Computer Science*, Vol. 641, pp. 312-18.

Zeglinski, G.W., Han, R.P.S. and Aitchison, P. (1994), "Object-oriented matrix classes for use in a finite element code using C++", *International Journal for Numerical Methods in Engineering*, Vol. 37, pp. 3921-37.

Zheng, Y., Lewis, R.W. and Gethin, D.T. (1995), "FEView – an interactive visualisation tool for finite elements", *Finite Elements in Analysis and Design*, Vol. 19 No. 4, pp. 261-94.

Zimmermann, T. and Eyheramendy, D. (1996), "Object-oriented finite elements 1. Principles of symbolic derivations and automatic programming", *Computer Methods in Applied Mechanics and Engineering*, Vol. 132, pp. 259-76.

Zimmermann, T., Dubois-Pelerin, Y. and Bomme, P. (1992), "Object-oriented finite element programming 1. Governing principles", *Computer Methods in Applied Mechanics and Engineering*, Vol. 98, pp. 291-303.