ELSEVIER

Advances in Engineering Software 30 (1999) 775–782

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

Object-oriented finite element programming-the importance of data modelling

R.I. Mackie*

Department of Civil Engineering, University of Dundee, Dundee DD1 4HN, UK

Received 1 October 1996; accepted 21 September 1998

Abstract

A C++ implementation of a finite element class system and its links to a graphical model of a structure are described. The principles underlying the finite element and graphical class systems are outlined, together with the reasoning behind the design. Two of the key points are (i) the finite element classes have a "lean" interface; (ii) the finite element objects (e.g. nodes and elements) are distributed around the graphical model objects (e.g. points, lines, sub-structures). Some of the advantages of adopting such an approach are outlined with reference to user interaction, mesh generation, and sub-structuring. © 1999 Elsevier Science Ltd and Civil-Comp Ltd. All rights reserved.

Keywords: Object-oriented programming; Finite elements; Object-oriented philosophy; Object-oriented code development

1. Object oriented programming and finite element software

Object oriented methods are all pervasive in computing. While most of the interest has centred on business type applications, databases and operating systems, there is an increasing level of interest in engineering software in general, and the field of finite element software in particular. The finite element method has been in use for some forty years now, and that means that there are millions of lines of FORTRAN code and billions of pounds invested in this code. Given this vast existing investment one needs to think very carefully before embarking on a radically different programming approach, and to be clear about what one is aiming to achieve, and how that goal is to be reached.

A number of articles on applying object oriented (OO) methods to FE have been published. These range from works which were straightforward implementations of FE in an OO language [1–3] to ones dealing with specific algorithms related to FE [4,5], or solving specific problems [6].

The fundamental problem in finite element programming is that of complexity. This complexity arises from several sources:

- The finite element method itself deals with a vast range of problems, covering all aspects of structures, and many in fluids, heat transfer etc.
- Some of the solution methods can be very complicated, especially in the areas of non-linear problems, sub-structuring and fluid-solid interaction.
- Many of the associated algorithms, such as mesh generation, are complex.
- An essential part of finite element programs is the graphical user interface, which adds further degrees of difficulty.
- There is a need to integrate finite element software with other software. Currently this is most pertinent to CAD systems, but the need for integration is likely to increase.
- There is a need to integrate the whole design, analysis and construction process. This will require the linking of FE programs to databases.

It is apparent from the above list that finite element programming is now about much more than merely implementing numerical algorithms (a task which can be difficult enough in its own right). This requires a much greater emphasis on data structures, and a greater ability to handle complex data. While these tasks can be handled with traditional languages, they are not really designed for this task. This is why there is indeed a need to consider new approaches, such as object oriented (OO) methods.

The following claims are made about object oriented programming [1,2]:

* Tel.: + 44 1382 344702; fax: + 44 1382 344816.
E-mail address: r.i.mackie@dundee.ac.uk (R.I. Mackie)

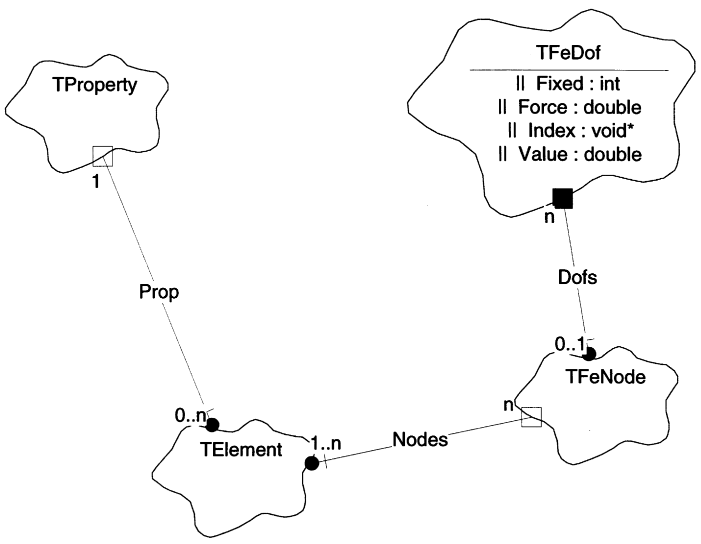

Fig. 1. Finite element class system.

- It enables the software to more closely model the real world.
- It produces more robust code.
- It encourages reuse of exiting code.

These work together to increase productivity. There is a degree of truth in all these claims, but OO does not yield these advantages automatically. In order for the benefits to be realised software design needs to be thoroughly thought out. In particular, there are a number of important design differences between the OO programs and FORTRAN ones.

This article will look at how these benefits can be realised. A case study of a C++ implementation of a two dimensional finite element program with a graphical user interface will be used to do this, but rather than merely describing the system, the article will focus on:

- The philosophy behind the overall design of the system
- Key elements in the design of class libraries to encourage and facilitate reuse.
- How the OO design yields advantages, both to the user and the software engineer, i.e., the article will propose guidelines and principles that, if followed, allow the benefits of an OO approach to be realised.

2. Object oriented design and finite elements

The system consists of two main models: a finite element class system, and a graphical–structural model. The key points are:

- The user interacts with the system entirely through the graphical model.
- The finite element class system is "used" by the graphical model, but the finite element class library was designed in such a way that it did not dictate how the graphical model should be constructed.
- Data was distributed around the graphical model, rather than being centralised

.

For example, the nodes belong either to key points, key lines or sub-structures of the graphical model. This means that the data structure more closely matches the way that a user views the data.

One of the reasons object oriented programs are more robust is that data is encapsulated within objects. This aids robustness in two ways:

(i) It allows control over access to data;
(ii) It means that changes to the data structure used within a class do not require changes to other parts of the program, providing the interface to the class has not changed.

However, this is only one aspect of robustness. Any software that is of significant size and is used over a period of time requires modifications, enhancements and error fixes. No matter how well designed a piece of software is, no one has perfect foresight. A common problem is that modifications to a program have unforeseen side effects, i.e., they produce new bugs. An approach that aims to increase software robustness must not merely make current code safer, but allow system to be more safely extended and modified. Therefore the main guiding principle behind the design of the finite element class system was that it must facilitate reuse and extension. Specifically:

- The class system itself must be able to be extended, e.g. extending a two dimensional system to a three dimensional one.
- The system must be able to be used in a wide variety of situations, such as in graphical models, links to CAD packages etc.

In order to achieve this end the class system must:

- Represent the underlying data structure of the basic finite element objects, e.g. degrees of freedom belonging to nodes etc.
- The classes must provide facilities to allow access to all necessary data and services.
- The class system must make no implicit or explicit assumptions about the data structures used in any system that might make use of the class system.

A fundamental difference is that the system is not designed with any particular application in view, rather a system is being designed that can be used by a wide variety of end applications.

2.1. The finite element class system

The base finite element system is shown in Fig. 1, using the Booch [7] notation. The system is very simple, and its prime function is to represent the underlying data structure of finite elements. It should be noted that, unlike some systems that were reported, there are no separate classes for applied loads or restraints. The precise nature of restraints or applied loads is dependent upon the particular application, and it is very difficult to introduce classes to deal with these at this stage without making assumptions about the eventual use of the system. Therefore restraints and loads are handled at the lowest possible level, namely by the *Fixed* and Force member fields in *TFeDof.* In a particular application higher level classes are used to convert the loads to equivalent nodal loads, and restraints are handled similarly. It should also be noted that while *TFeNode* class owns one or more *TFeDof's*, it is also possible for *TFeDof's* to be free standing, as might be needed in handling more complex restraints. The *Index* field of *TFeDof* is used for indexing purposes in, for example, solution routines. The definitions of *TFeNode* and *TElement* represent the fact the nodes own degrees of freedom, and that elements have nodes associated with them. *TElement* also has pure virtual functions to represent the fact that elements have stiffness and mass matrices. *TProperty* is completely undefined at this stage, it will typically represent such things as material properties of elements.

The classes have a minimal interface, e.g. the class *TElement* has the following interface:

```
class TElement {
public:
TElement(int noNodes, TFeNode**nodes, TProperty*-
prop);
Virtual ~ TElement( );
int GetNoNodes( );
int GetNoDofs( );
virtual int CalcStiffMat(unsigned intOrder, int&, noDofs,
double*k) = 0;
…other functions that calculate mass matrices etc.
};
```

It might be thought that other functions such as ones to access the values of the degrees of freedom associated with the element would be needed. This is not necessary. The difficulty with this approach is that it can lead to a proliferation of functions, and even then it cannot be guaranteed that all the user's needs will be met. An alternative solution, and the one used here, is to use iterators. For instance an iterator *TElemDofIter* is defined that automatically runs through the elements degrees of freedom. If *elem* is an instance of *TElement,* then it is used as follows:

```
TElemDofIter iDof(elem);
while (iDof) {
TFeDof dof = iDof++;
… do something with dof
};
```

The first line initialises the iterator, the following code then runs through each of the degrees of freedom in turn. The iterator allows access to the elements degrees of freedom without making any assumptions about what the program may wish to do with them. Iterators *TNodeDofIter* and *TelemNodeIter* are also defined to allow access to a nodes degrees of freedom, and elements nodes, respectively.

In working applications various descendants of *TFeNode* and *TElement* will be used. If *TelemNodeIter* is used, then frequent type casting is needed, this is both a nuisance and a potential source of errors. A better alternative is to use a template class:

```
template < class TElem, class TNode > class TEle-
mNodeIterX: public TElemNodeIter { public
TElemNodeIterX(Telem*elem);
TNode*Current();
TNode* operator ++(int);
TNode* operator ++();
};
```

So if for plane stress problems elements *TElemenPS* and *TNodePS* were defined, then the typedef

```
typedef TElemNodeIterX < TElementPS,TNodePS >
TElemNodePSIter;
```

declares an iterator that is initialised with an element of type *TElementPS* and returns pointers to nodes of type *TNodePS.* This is a neater solution that the repeated use of typecasts, and other iterators can be defined for other combinations of elements and nodes.

Further details of the finite element class system can be found in Mackie [8]. As it stands the system cannot be used directly, rather it provides a compact, yet solid foundation from which working class systems can be derived.

2.2. Extension to plane stress

In order to develop useable element for plane stress problems the base class system needs to be extended in two ways. First it needs to be extended to deal with two

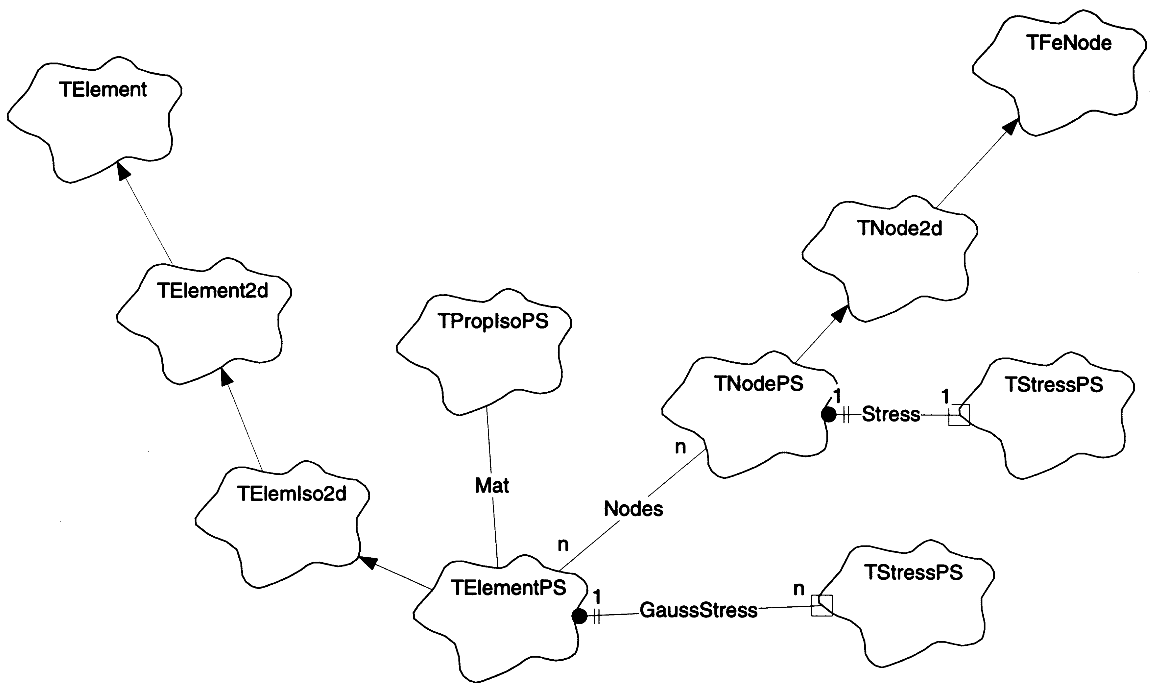

Fig. 2. Derivation plane stress classes.

dimensional problems, and then specifically with plane stress problems. The class hierarchy is shown in Fig. 2.

For the node classes, *TNode2d* is derived from *TFeNode* with the additional data member being its location in two dimensional space. *TNodePS* is then derived from *TNode2d.* This has one new data member, *TStressPS.* TStressPS contains the three stress functions, and also has functions to give the corresponding principle stresses. Other features of the class are that (i) it knows that it has two degrees of freedom, the *x* and y displacements; and (ii) the node has stress values. Functions *GetU( )* and GetV*( )* are added to return the *x* and y displacements. Other access functions are provided to deal with loads and restraints associated with the node in a meaningful way, e.g. they "know" that the load has *x* and y components. The base class *TFeNode* stored the degree of freedom information in a sensible, but non-intuitive manner, *TNodePS* gives meaning to these degrees of freedom, i.e., *TFeNode* provides the data structure. *TNodePS* provides the interpretation. The extension to two dimensions involves two classes. The first *TElement2d* has no new data members, but knows that all its nodes are of class *TNode2d.* TElemIso2d also has no new data members, but represents fully two dimensional elements, and has member functions that deal with various calculations associated with iso-parametric elements. Elements such as plane frame or truss elements would be derived from *TElement2d,* whereas elements such as plane stress elements are derived from *TElemIso2d. TElementPS* has additional data members to store the stress information at the gaussian integration points. The property class *TPropIsoPS* contains the thickness and material properties of the elements. The functions such as *CalcStiffMat* are now property defined, so *TElementPS* is a fully working element.

The above describes the essential details of the plane stress system, fuller details can be found elsewhere [8].

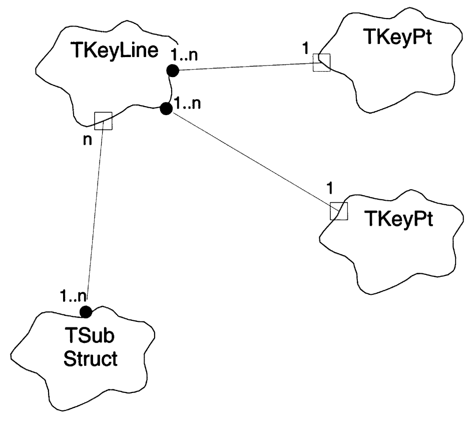

Fig. 3. Main graphical classes.

3. Linking the FE model with the graphical model

This section will describe how the FE system can be used in a graphical model. The graphical model consists of three main classes: *TKeyPt, TKeyLine,* and TSubStruct. A key point is a point in space. A key line has two key points at its beginning and end, the line itself may be straight or curved. A sub-structure is a closed region consisting of several key lines. The class relationships are shown in

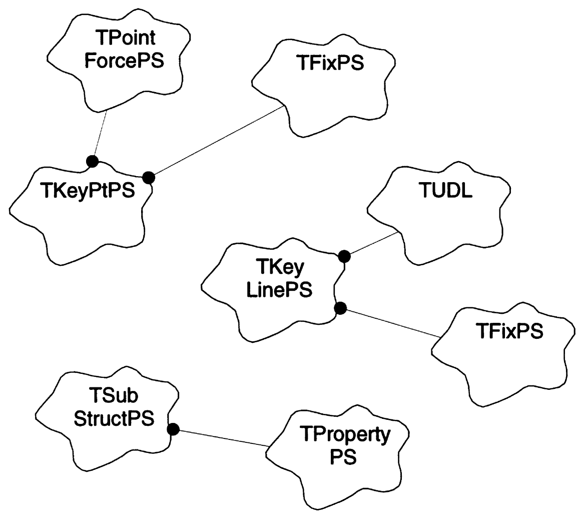

Fig. 4. Graphical classes for plane stress.

Fig. 3. A complete model will consist of one or more sub-structures. Further classes are derived from these for particular applications. For a plane stress program, classes *TKeyPtPS*, *TKeyLinePS*, and TSubStructPS are derived, together with classes for point force, distributed loads, and various types of restraint, and the ownership of these is shown in Fig. 4. A simple example of what the user sees is shown in Fig. 5. As far as the user is concerned the point forces, and other entities, belong to points, or lines, and this is the way that the object model represents the data, i.e., the software views the data in much the same way as the user does, this has significant advantages, as will be outlined later in the article.

It is common in Fortran programs to store the data centrally, e.g. there may be large lists or arrays of nodes and elements. In the current system the data is distributed. Fig. 6 shows how nodes belong to key points, key lines and sub-structures, and elements belong to sub-structures.

When the mesh is generated the nodes along each of the key lines belong to that line, or, in the case of the ends of the line, to the key point. The other nodes belong to the relevant sub-structure, as do the elements. The restraints and applied

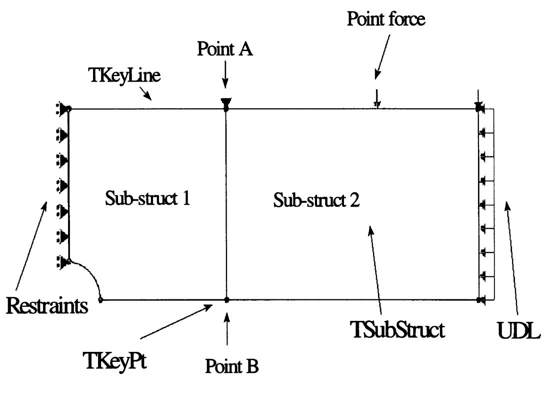

Fig. 5. Simple example of geometric model.

loads present in the graphical model are converted into equivalent nodal loads or restraints. E.g. the uniformly distributed load acting along a line is converted to equivalent nodal loads and applied to the nodes belonging to that line, similarly if x restraints are applied along a line, then the degree of freedom associated with x displacement for each node associated with that line is restrained.

4. Advantages of the distributed data approach

There are a number of advantage to this distributed data approach: for the user, in terms of software development, and in terms of algorithm implementation. Three examples are briefly described below:

4.1. User access to data

The way that the data is stored within the model much more closely matches the way that the user views the data. For instance, in the example shown in Fig. 5 it is the results around the arc of the hole that are of most interest to the user. To access information on this arc (or any other line, for that matter), all the user needs to do is right click on the line and a menu will appear. The user can then select, say, the stress option, and a table or graph showing the stresses around the arc will appear. Now, this could be implemented in any language, but with the object-oriented approach the nodes holding the relevant information is held are owned by the arc, i.e., the information is readily available, and no complex indexing operations are needed to determine the relevant nodes.

4.2. Mesh generation

The data ownership offers a number of advantages in terms of automatic mesh generation. Suppose a model has several sub-structures and the mesh has already been generated. Now if the model is modified in some way, then rather that regenerating the whole mesh, only the sub-structures affected by the change need to have their elements and nodes re-generated. For a small model this is not a major saving, but in more complex cases can be quite significant.

The generating process itself is also helped by the object-oriented approach. The mesh is generated separately for each sub-structure. Starting with the nodes on the boundary of that sub-structure, an advancing front scheme (Blacker and Stephenson [9]) is used to generate the other nodes and elements. These boundary nodes are taken from key lines comprising the boundary of the sub-structure. Now if a key line is part of two sub-structures, then the nodes belonging to that line will belong to the elements that form the interface between the two sub-structures. Fig. 7 shows the object ownership and access diagram for sub-structures 1 and 2, line AB, and key points A and B of Fig. 7.

Since line AB is part of both sub-structures, the nodes belonging to AB are also part of the two sub-structures, thus

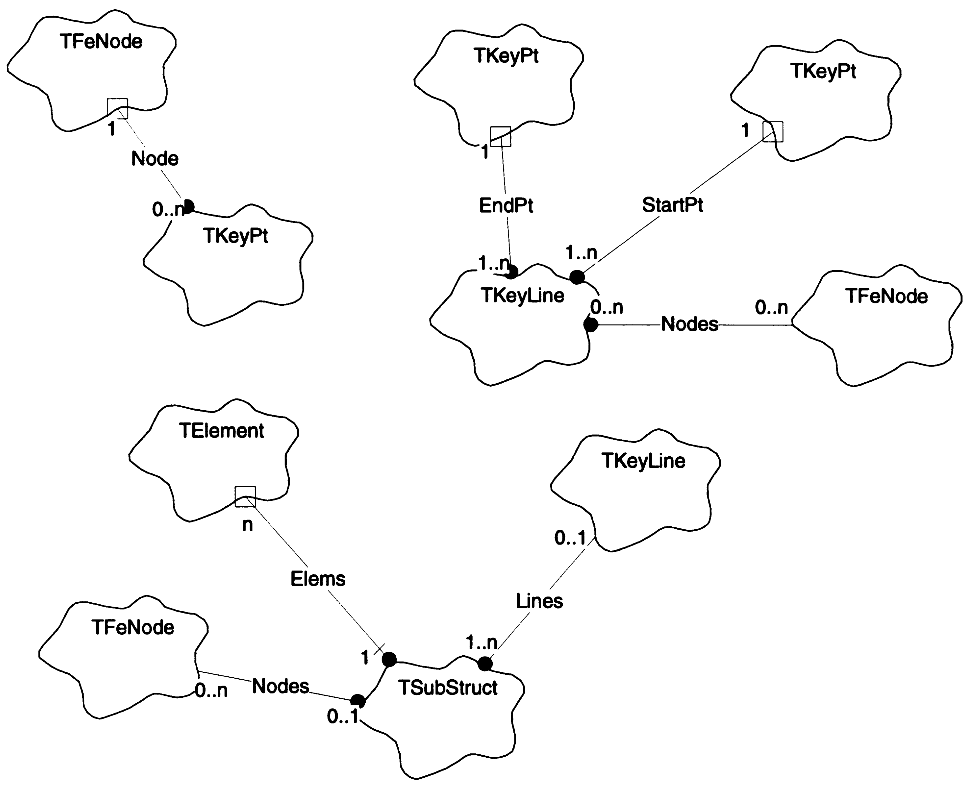

Fig. 6. How finite element objects are owned by graphical objects.

mesh continuity is assured. This happens as a result of the data structures used, not as a results of direct programmer intervention.

Sometimes mesh continuity is not wanted, such as in Fig. 8 where AB is a crack. In this case there would be two key lines AB_1 and AB_2. The key point A would be common to sub-structures 1 and 2, so the node belonging to A would also be common. B_1 and B_2 would be geometrically coincident, but would have different nodes, also the nodes belonging to AB_1 and AB_2 would be different. Therefore when the mesh for sub-structure 1 was generated it would use nodes from A, AB_1 and B_1, whereas the mesh for sub-structure 2 would use nodes from A, AB_2 and B_2, and the fracture would be represented properly. Again, ensuring that this happens is a result of a high-level decision in the data structure used, not a low-level programming decision. The node ownership is shown in Fig. 9.

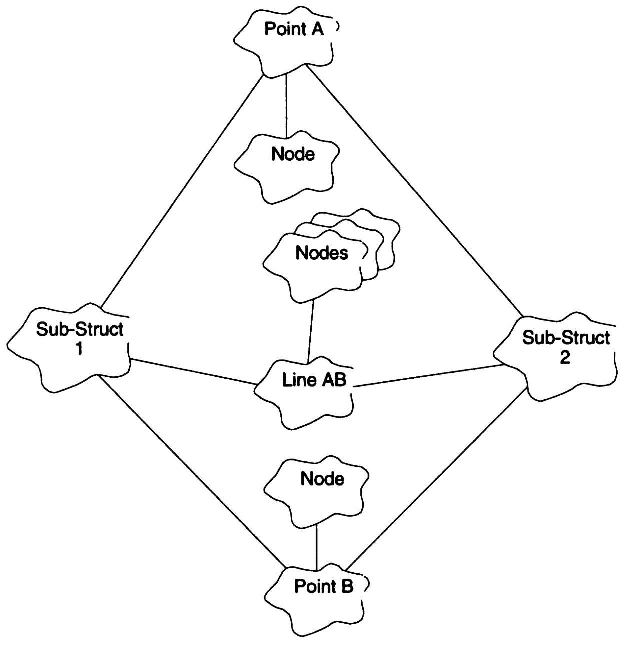

Fig. 7. Node ownership diagram for Fig. 1.

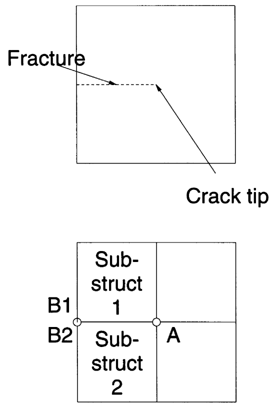

Fig. 8. Structure with fracture line.

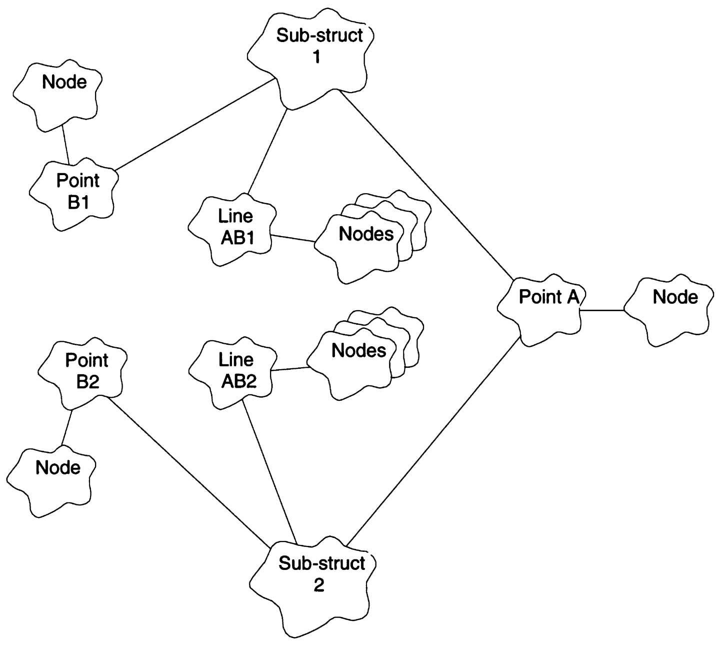

Fig. 9. Node ownership for line fracture line in Fig. 1.

In addition to these advantages, the detailed programming of mesh generation, and other algorithms such as node or element renumbering schemes, are helped by an object oriented approach. This is because of the richer data structures which are available.

4.3. Sub-structuring

The sub-structuring method was originally developed in order to reduce the memory requirements of finite element programs. The method works by obtaining a reduced stiffness matrix for a sub-structure by eliminating the degrees of freedom associated with internal nodes. This leaves a stiffness matrix and load vector for the sub-structure expressed in terms of the boundary nodes alone. The reduced stiffness matrices for the various sub-structures can then be assembled and solved as with normal elements. An object oriented scheme for doing this was presented by Ju and Hosain [5]. In traditional programming languages the main problem with implementing sub-structuring is one of book keeping. The object oriented approach makes this task much simpler, because elements and nodes are owned by sub-structures, lines etc.

Memory available on computers is now much greater than it used to be, and so memory restraints are not as severe as they once were. However the method is still relevant from a computational point of view:

1. Sometimes extremely large problems are required to be solved.
2. As computer speed increases the scope for more interactive, rather than batch mode, solution becomes an increasing possibility. The use of sub-structuring could help with this. E.g. the user may wish to alter part of a structure and study the effects, sub-structuring would facilitate this.
3. On a management level it might be desirable to split a particular design problem into several parts. A problem then is bringing the information together again so that the whole problem can be looked at. The difficulties here are ones of data management, and the richer data modelling capabilities will be of significant benefit.

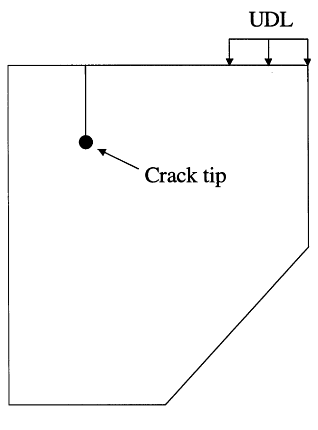

Fig. 10. Sample problem.

4.4. Example application

Several finite element programs were written using the principles outlined in this article. It is useful in developing programs for undergraduates. The object oriented technique enables programs to be developed that are both user-friendly and powerful. This allows students to tackle problems of moderate complexity and size (e.g. several thousand nodes). A typical problem is illustrated in Fig. 10. It consists of a bracket with a crack in it, and the task is to calculate the stress intensity factor at the crack tip using quarter-point elements around the crack-tip. Tackling this problem makes use of several of the features outlined in this article, namely:

- The general ease of use of the program makes it relatively easy to design the mesh. The sub-structure based nature of the underlying object model means that it is possible for the meshes to be generated one sub-structure at a time. Therefore attention can be focussed on one sub-structure at a time when designing the mesh.
- The correct inter sub-structure mesh continuity is automatically ensured because of the features described in Section 4.2. In particular the mesh around the crack is handled correctly (see Figs. 8 and 9).
- In order to ensure that quarter point elements are used around the crack tip, the user merely needs to set the appropriate property on the key point located at the crack tip. The correct mesh is then generated.
- In order to use the crack-opening displacement formula to calculate the stress intensity factor, the user needs the displacements along the crack. This information is easily obtained by clicking on the crack.

Therefore the features outlined in the foregoing sections make the solution of the problem straightforward.

5. Conclusions and discussion

- The demands on modern engineering computing are becoming ever more complex. This is placing a much greater premium on data modelling, in addition to implementing numerical algorithms. Object oriented methods are well suited to this challenge.
- The successful application of object oriented methods to engineering software requires a very different approach to software engineering than is used in traditional scientific programming. A key element of this new approach is emphasis on representing data structures.
- Some of the advantages of adopting an object oriented approach in terms of user interface, algorithm implementation, and software design are presented in this article.

Finally, it is important to remember that a vast amount of money, time and effort have already been invested in finite element software. This naturally raises the question of whether or not it is feasible or desirable to apply a radically different approach. There are strong reasons for giving the answer yes. However, the options are not quite as stark as they may first appear. While the application described herein is a complete finite element system, from user input through calculations to final output, this is not the only way that objects can be use. Instead of producing a finite element model from the graphical model, and then solving it, the graphical model could just as easily be designed to produce a data file that could be used by existing software, i.e., object models could be used to handle the aspects of problems where data handling is the key issue, and then revert to existing finite element codes for the solutions routines. Such an approach is not ideal and would not yield the full advantages of the object approach, but it could form a useful half way house.

References

[1] Zimmermann T, Dubois-Pelerin Y, Bomme P. Object-oriented finite element programming: I. Governing principles. Computer Methods in Applied Mechanics and Engineering 1992;98:291–303.

[2] Forde BWR, Foschi RO, Stiemer SF. Object-oriented finite element Aanalysis. Computers and Structures 1990;34:355–374.

[3] Mackie RI. Object-oriented programming of the finite element method. Int J Num Meth Engng 1992;35:425–436.

[4] Fenves GL. Object-oriented programming for engineering software development. Engineering with Computers 1990;6:1–15.

[5] Ju J, Hosain MU. Substructuring using the object-oriented approach. In: Topping BHV, Papadrakakis M, editors. Proc 2nd Int Conf on Computational Structures Technology, Athens, 30th Aug–1st Sept, Civil-Comp Press, 1994. pp. 115.

[6] de Oliveira e Sousa JLA, Ingraffea AR. An object-oriented system for the 3-dimensional simulation of hydraulic fracturing processes in rock. Proc 2nd Int Conf on Computational Structures Technology 1994 pp. 139–146.

[7] Booch G. Object Oriented Analysis and Design with Applications. In: Cummings B, editor. 2nd ed. CA: Redwood City, 1991.

[8] Mackie RI. Using objects to handle complexity in finite element software. Engineering with Computers 1997;13:99–111.

[9] Blacker TD, Stephenson MB. Paving: a new approach to automated quadrilateral mesh generation. Int J Num Meth Engng 1991;32:811–847.