Computer Assisted Mechanics and Engineering Sciences, 3: 65–74, 1996.
Copyright © 1996 by Polska Akademia Nauk

A prototype object-oriented finite element method program: Class hierarchy and graphic user interface[1]

R. Robert Gajewski and Tomasz Kowalczyk
Warsaw University of Technology, Faculty of Civil Engineering,
Center of Computer Methods, 00-637 Warszawa, Armii Ludowej 16, Poland

(Received August 7, 1995)

The paper considers the application of an object-oriented approach to the development of FEM software. A brief introduction to basic concepts of object-oriented modelling is given, followed by a short overview of developed classes. Objects, classes, methods and inheritance are illustrated using a graphical representation. The design, implementation and maintenance of an object-oriented program is compared to that of an equivalent procedural program in order to identify advantages of the object-oriented approach. Some design problems of conventional finite element analysis software and their possible solutions offered by the object-oriented methodology are identified and discussed.

1. Introduction

All large software systems tend to be changed and to evolve over time. When we correct errors in them we speak about software *maintenance* and we speak about *evolution* when we respond to changing requirements. We can also speak about *preservation* when we use extraordinary means to keep an old piece of software in operation. Investments in a large program are spent mostly during its evolution. Due to the increasing size and complexity of finite element software systems the traditional algorithm-driven structured programming approach and, in majority of the cases, FORTRAN as the programming language are no longer adequate. New software architecture and improved programming paradigms supporting maintainability, which can be defined as a combination of three qualities: understandability, extendability and easy debugging, should be created. In this paper an object oriented approach to scientific programming in the field of finite element analysis is discussed.

All industrial-strength software systems involve elements of great complexity. Only small applications created by the amateur programmer or the professional developer working in isolation are not complex. The complexity of software is its essential property, not an accidental one. Today it is not unusual to find finite element method systems whose size is measured in hundreds of thousands lines of code written in a high-order programming language. Even if we decompose such implementation in some meaningful way we still end up with hundreds of separate modules. Our failure to master the complexity of software results in the so called software crisis. Moreover, the more complex the system, the more open it is to total breakdown.

When designing a complex software system it is essential to decompose it into smaller parts each of which can then be refined independently. Majority of the programmers have been formally trained to use the method of top-down structured design. Decomposition is approached as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some

[1]The paper was presented at the 12th Polish Conference on Computer Methods in Mechanics in Warsaw–Zegrze on May 9–13, 1995. Further papers presented at the Conference will be published in the forthcoming issues of this Journal.

overall process. There is an alternative decomposition possible for the same problem. The system can be decomposed according to key abstractions in the problem domain. In this decomposition, we view the world as a set of individual agents that collaborate to perform some higher level behaviour. Because such decomposition is not based upon algorithms but upon objects, which embody unique data and behaviour and model some entities from the real world or mathematical model, we call this an object-oriented decomposition.

The need of alternative approaches to FEM software is generally recognised. Papers on usage of C language in finite element programming (see [4, 10]) and computer programs written in C appeared recently. In recent years many papers on the usage of object-oriented techniques in finite element programming appeared as well (see, e.g. [3, 5, 6, 15]). This methodology is also gaining interest in Poland (see, e.g. [7, 8, 13]).

2. Object modelling

Object-oriented technology is built upon a foundation whose elements we collectively call the object model. The object model encompasses the principles of abstraction, modularity, hierarchy, typing, concurrency and persistence. None of these principles are entirely new, but what is important about object model is that these elements are brought together in a synergistic way.

Without doubt, object-oriented modelling consisting of analysis and design is fundamentally different from traditional structured design approach. It requires quite different way of thinking about decomposition and it produces software architectures that are largely outside the realm of the structured design culture. The complex systems we deal with have a hierarchical nature. The levels of this hierarchy represent different levels of abstraction — each of them is built upon the other and each is understandable by itself.

In the common opinion of the specialists in the field of software engineering, object oriented approach is more suitable there than procedural one, because it is better at helping us organise the inherent complexity of software systems.

2.1. Basic terms and concepts

An object is a computer analogue of entities in the real world or mathematical models which consists of an encapsulated representation (state) and a set of messages (operations, procedures) that can be applied to the object. In other words, they are capsules of behaviour and state whose internals are hidden from other objects that use their services. Only the methods of an object have an access to its state. A method can only be invoked by sending the object a message. Instead of organising programs into procedures that share global data, the data is packaged with the functions that access the data.

A class is a collection of objects with common attributes and/or behaviour. An object is an instance of a class — one of the things in the class (see Fig. 1).

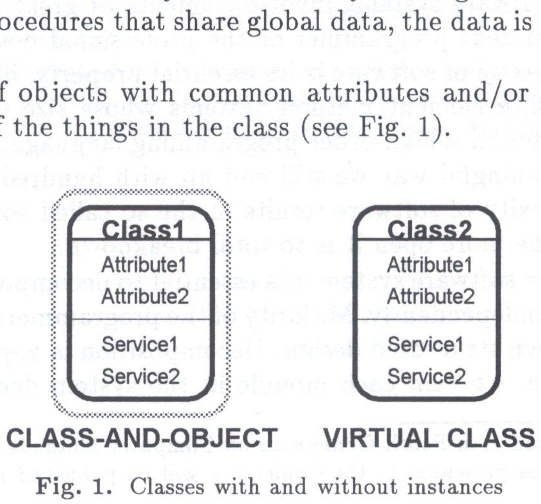


Fig. 1. Classes with and without instances

Inheritance is an ability to define a new class that is just like an old one except for a few differences. Classes may share in this way their common attributes and methods.

Polymorphism is an ability of different objects to respond differently to the same message.

All these features can potentially improve the limitations of procedural programming. The decomposition performed in terms of objects better parallels the real-world problems or mathematical models. The concept of classes provides mechanisms to reduce the contact surfaces between software modules. The mechanism of inheritance allows the reuse of software. The concept of polymorphism enables the construction of programs on a higher level of abstraction.

2.2. Object oriented analysis and design

One of the main questions confronting a software developer is: Which is the right way to decompose a complex engineering software system — by algorithms or by objects? The right answer is that both views are important. However, we cannot construct a complex system in both ways simultaneously. It is due to the fact that these views are orthogonal, representing active and static points of view which are dual by their nature. Experiences of many researchers led the authors to apply the object-oriented view.

Despite individual differences in all object-oriented system development methodologies, they always contain three components corresponding to analysis, design and implementation/programming.

Analysis involves problem definition and modelling (see [1] or [2] for more details). Object-oriented analysis models the problem domain by identifying and specifying a set of semantic objects which represent things or concepts rather than a solution method. They are called semantic objects because they have meaning in the problem domain.

Design focuses on solution specification and modelling. Object oriented design transforms the problem representation into solution representation. The solution domain includes semantic classes with possible additions and interface, application and base/utility objects defined in the design process. During the design phase the emphasis is on defining a solution. Object-oriented design should be still language-independent. It precedes physical design (implementation, programming).

The most important part of problem domain analysis process is connected with identification of attributes, behaviour and classes and final organisation of them. Attributes and behaviour must be distributed among appropriate classes. Then classes are arranged into predefined relationships or organisations.

The two primary ways of organising objects, or two kinds of generic relationships among objects are (see Fig. 2):

- *a-kind-of* structure (generalization–specialization),
- *a-part-of* structure (whole–part, aggregation).

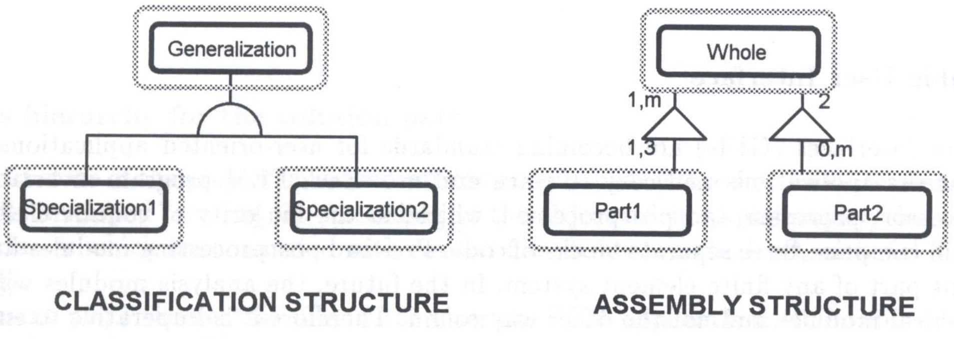


Fig. 2. Basic static relationships between objects

A-kind-of structure uses inheritance, which is the major feature distinguishing the object approach from other approaches. It is used as a mechanism to create objects that share properties with similar objects.

Inheritance is not the only way to share code and promote reuse. A-part-of structure uses parts which enable the construction of compound objects such as windows system in the graphic user interface. They are used as a mechanism for assembling composite objects. In the description of composite objects it is necessary to outline how they are created and how components are added and modified.

In the assembly structure, objects of the class *Whole* contain exactly two objects of the class *Part2* and at least one object of the class *Part1*. On the other hand, object of the class *Part1* is a part of at least one and at most three objects of the class *Whole*. Finally, object of the class *Part2* can be a part of any number of objects of the class *Whole*.

Figure 2 addresses only structural (static) aspects of object-oriented analysis and design, i.e., the object model. There is also a dynamic model which describes the aspects of a system that change over time. It is used to specify and implement the control aspects of the system. Such dynamic considerations, namely message-passing between objects, are presented on Fig. 3. The third model, functional model, used to describe the data value transformations within the system will not be discussed here.

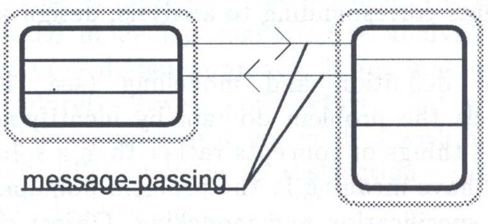


Fig. 3. Dynamic aspects of object-oriented analysis and design

3. Object Oriented Finite Element Programming

Nearly all FEM software is based on computational methods unchanged since the time when the batch mode of computer data processing was the norm (see, e.g. [11, 14]). FORTRAN dominates in the field of scientific computing both in commercial packages and university programs. This is mainly due to the assumption that structural analysis is performed on well defined and static data.

A typical FEM code consists of a processor hidden from the user by pre- and postprocessing software. The analysis is mainly performed in batch mode. This approach has its natural limitations. Complex control and artificial data structures result in difficulties of development and maintenance of such conventional software.

3.1. Graphic User Interface

Graphic User Interfaces (GUIs) are becoming standards for user-oriented applications. They are consistent across applications and easy to learn and use. Each FEM program invariably consists of a preprocessor, processor and postprocessor, which, in the majority of commercially available packages, still comprise three separate blocks of code. Pre- and postprocessing modules have become an important part of any finite element system. In the future, the analysis modules will rather be part of graphical modules and not the other way round. Therefore it is imperative to employ novel software structures if these objectives are to be met.

While designing an application it is reasonable to separate the part of the system responsible for calculations, i.e. the solver, from the GUI classes to make them independent and thus more flexible. We encounter here problems with establishing proper relations between these two modules.

The essence of a GUI are *GraphicObjects.* These objects posses two important abilities. First, they can display themselves on the specialised *ProjectionWindow,* and next, they can communicate with the user (e.g. through the dialogue box), to let him change their characteristic features.

It is obvious that some of the *GraphicObjects* should represent *Nodes, Elements* and *Loads* which are the part of the solver and thus, as the objects responsible for calculations, they should not make assumptions about the graphics.

There are generally two ways of designing *GraphicObjects.* In the first approach, full independence of the solver from the graphic environment is assumed. The solver and its classes, written in the purely portable code, are completely separated from the rest of the system. *GraphicObjects* are not derived from the objects they represent. They are only *Handles* to them. Additional messages between the modules are necessary — from the *Handles* to the objects they represent — in order to retrieve their data.

In the second solution, the solver and the graphic environment are connected through the same objects they operate on. *Nodes, Elements* and *Loads* are themselves *GraphicObjects.* Thus both the solver and the graphic environment can operate directly and independently on the same objects. It is important that we can still logically separate the two parts of the classes: responsible for calculations and responsible for graphic representation. No additional messages between the two modules are necessary here.

Comparison of the two possible solutions is summarised in Fig. 4.

Due to the lack of fully portable graphic library, the first solution in which the role separation is assumed was chosen in our project [12]. The proposed class hierarchy of such a GUI is depicted in Fig. 5.

Separation	**Integration**
Handles	The same objects
– Introduction of additional, unnatural idea of *Handles,* which has not its counterpart in OO Analysis and is outside the Problem Domain.	+ Clarity of the concept of "both calculated and visible" objects. Easy development of object's specialized features.
+ Full portability of the solver.	– The whole code is not portable as long as we do not have the portable graphic library.
Graphics depends on the platform and must be written separately for different ones.	Graphics is written once, but its portability depends on the availability of the same graphic library on the other platform.

Fig. 4. Comparison between two ideas of GUI

3.2. Class hierarchy for the solution part

In the current state of the project, a revised version of a class structure for the finite element analysis program was developed (see Fig. 6). According to the object-oriented philosophy, the kernel of it is formed by classes which are connected with entities like element, node, load, material and structure (see also [9]).

The element plays the central role in the classical FEM applications. It includes all necessary information about the set of partial differential equations that will be solved. In the object-oriented

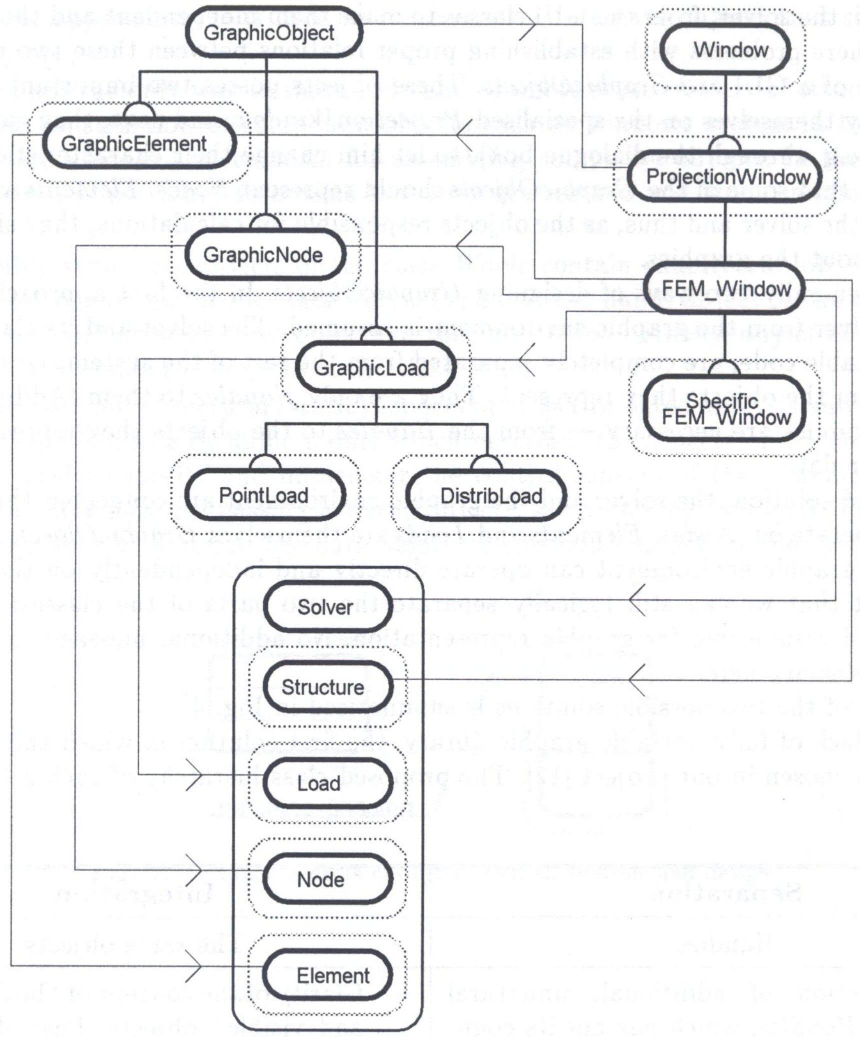


Fig. 5. Class hierarchy for graphic user interface

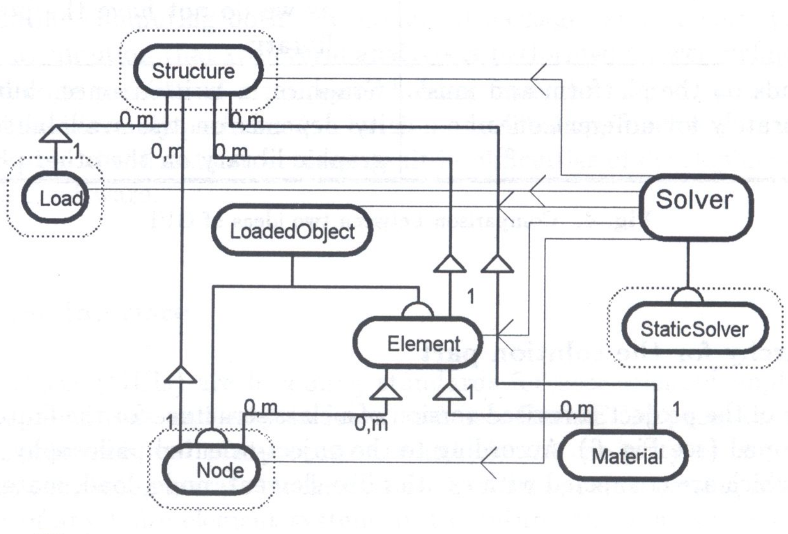


Fig. 6. Class hierarchy for the solution part of the FEM program

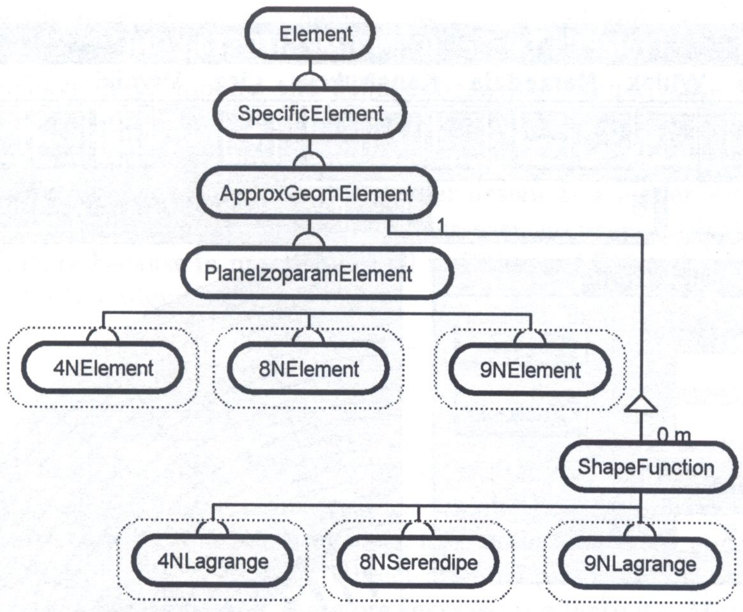


Fig. 7. Class *Element*

approach it has the same key role. The virtual class *Element* is designed to perform such tasks as computing the stiffness matrix and assembling the contributions to the linear system. The element also knows how to create itself and how to identify its attributes like nodes or material.

Structure of *Element* class is shown in Fig. 7. Any specific element performs similar actions, like calculations of stiffness matrix. The algorithm to carry out the action will differ depending on element's characteristics. Therefore elements are defined through inheritance.

This structure allows for straightforward inclusion of new elements. Each element should only have information about topology, connectivity and material properties. Each element object is only responsible for shape function calculations and numerical integration. Physical model of the problem is represented by the class *Material*. It can not only define material constants but also more complicated constitutive relations.

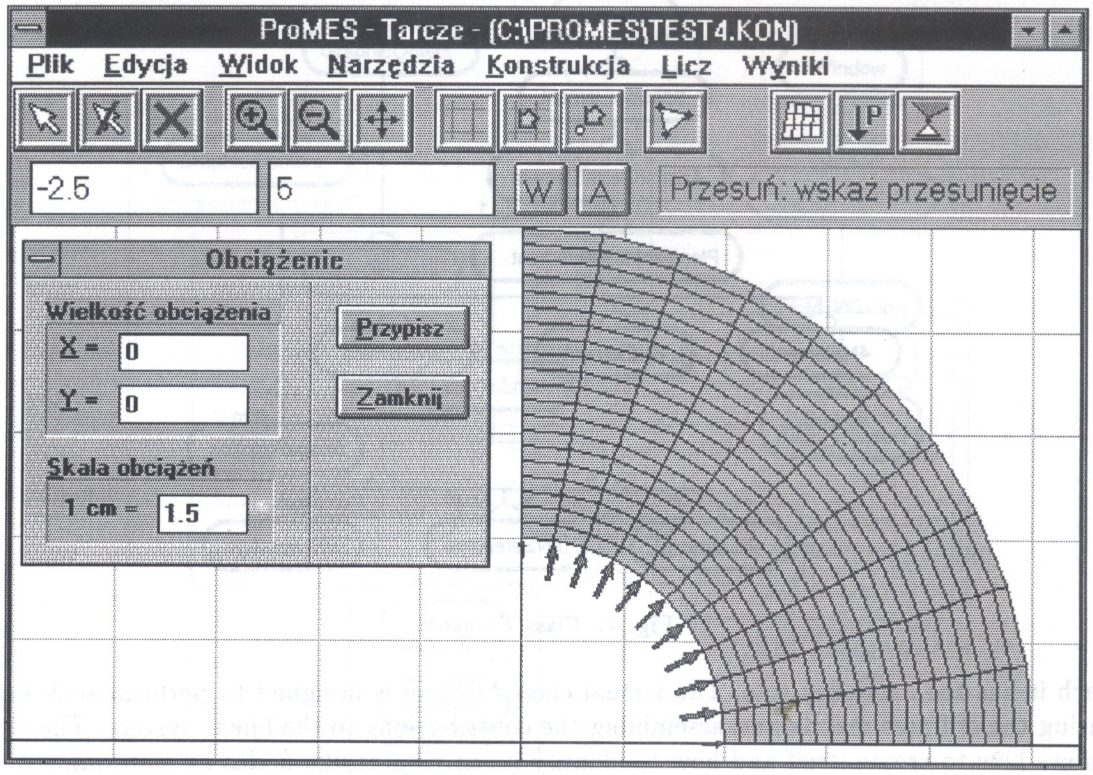
3.3. Implementation

The results of the performed object-oriented analysis and design have been coded and implemented in the C++ language, a C extension for object-oriented programming. The program PRO_MES was developed for Microsoft Windows GUI and DOS operating system (see Fig. 8). The Borland C++ compiler was chosen for the project development. It supports the concept of object-oriented programming by expanding the environment using Object Window Library, which permits the reuse of classes already defined. Windows environment gives many benefits like a device-independent graphics, support for a wide range of printers, monitors and pointing devices, a rich library of graphical routines, more memory than plain DOS for large programs and support for menus and icons.

The prototype program was tested on many benchmark problems from theory of elasticity. Sample results of calculations for the problem from Fig. 8 are depicted in Fig. 9.

3.4. Benefits of object-oriented approach

Encapsulation and information hiding reduce the contact area between the objects and localise the implementation details. Data structure and some (private) methods are hidden and can only be accessed through the public methods forming object's interface.



-2.5 5

Przesuń: wskaż przesunięcie

Wielkość obciążenia

X = 0

Y = 0

Skala obciążeń

1 cm = 1.5

Fig. 8. Graphic User Interface for FEM calculations

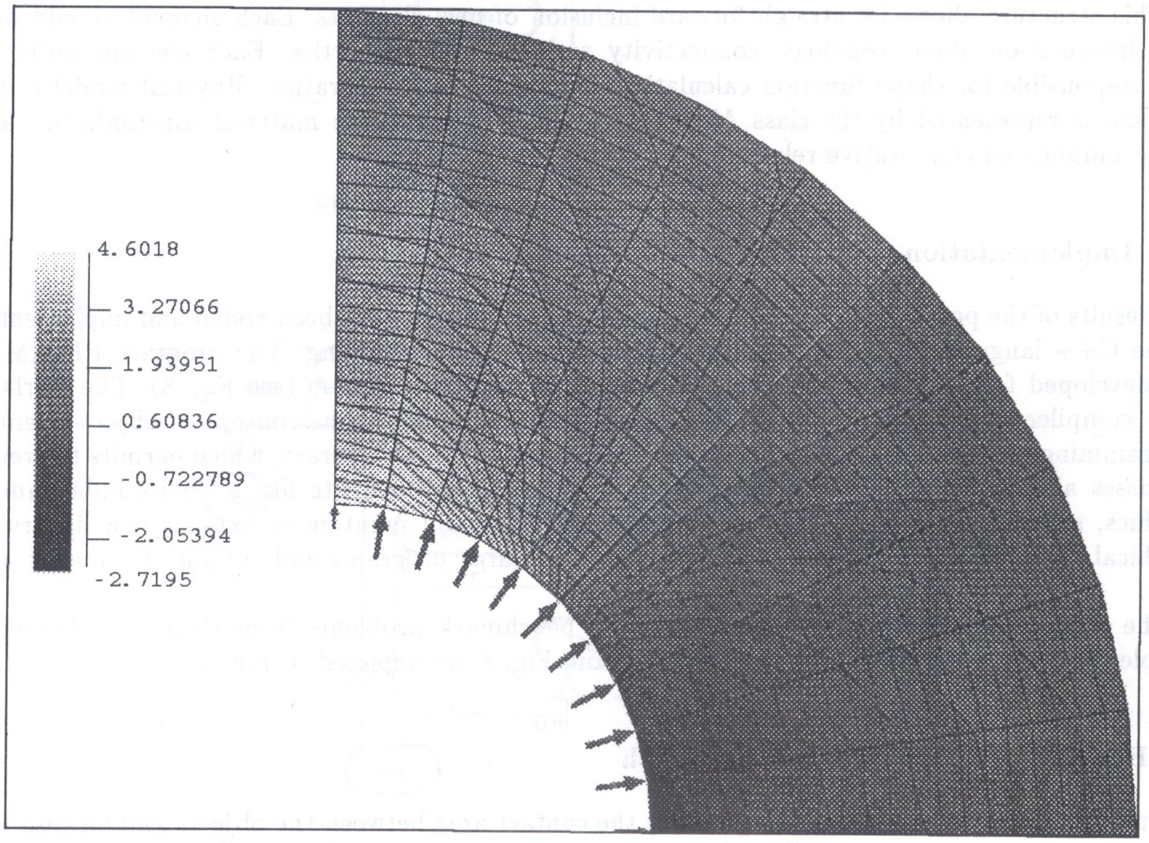


Fig. 9. σ_{11} stress

With inheritance the programmer can define a new class and add data items or methods as necessary. The derived class inherits all attributes of the parent class. The added data and methods either create new behaviour or modify the defined behaviour. Inheritance encourages software re-use, reduces the code size and finally enhances the flexibility of programming.

Polymorphic operations, having multiple meanings depending on the type of the object they operate on, allow a programmer to implement a cleaner design at a higher level of abstraction.

Last but not least, object-oriented approach enables easy program evolvability. This seems to be very important feature because in practice not all facilities of FEM code could be fully foreseen at the time it was initially designed.

4. Conclusions

The results obtained from the prototype program show many advantages of an object-oriented approach. It can solve all the longstanding problems with the development of engineering software systems, e.g. finite element analysis programs, like: maintainability, understandability, readability, homogeneity, extendability, efficiency. Object-oriented programming enables a programmer to create very high level operations and therefore increase productivity. Moreover, with this approach prototyping of a software system can be fast and inexpensive. The general conclusion is that object-oriented programming is a powerful tool and can be easily used to improve the quality of engineering software. In contrast to procedural programming, object-oriented programming results in smaller programs, provides better management of data and procedures, and makes easy extension of the program possible.

Acknowledgements

Partial financial support received under a grant from the Rector of Warsaw University of Technology is gratefully acknowledged. The first author also wishes to thank Monica for her inspiration, helpful comments, assistance and criticism.

References

[1] G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, 1994.

[2] P. Coad, E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood Cliffs, 1990.

[3] Y. Dubois-Pellerin, Th. Zimmermann. Object-oriented finite element programming: III. An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering,* **108**: 165–183, 1993.

[4] P. Fazio, K. Gavri. Structural analysis software and the C programming language. *Computers and Structures,* **25**: 463–465, 1987.

[5] J.S.R.A. Filho, P.R.B. Devloo. Object-oriented programming in scientific computations: the beginning of a new era. *Engineering Computations,* **4**: 81–87, 1991.

[6] B.W.R. Forde, R.B. Foshi, S.F. Stiemer. Object-oriented finite element analysis. *Computers and Structures,* **34**: 355–374, 1990.

[7] R.R. Gajewski. An object oriented approach to finite element programming. In: B.H.V. Topping, M. Papadrakakis, *Artificial Intelligence and Object Oriented Approaches for Structural Engineering,* 107–113. Civil-Comp Press, 1994.

[8] R.R. Gajewski, T. Kowalczyk. Object-oriented finite element programming: new stage or curiosity?. *Computer Methods in Civil Engineering,* **4**: 71–82, 1994 (in Polish).

[9] R.R. Gajewski, T. Kowalczyk, S.A.S. Zieliński. Class hierarchy for the prototype FEM program. In: *Proceedings of the XII Polish Conference on Computer Methods in Mechanics,* 108–109. Military University of Technology, Warsaw, 1995.

[10] K.H. Ha. C language for finite element programming. *Computers and Structures,* **37**: 873–880, 1990.

[11] E. Hinton, D.R.J. Owen. *Finite Element Programming.* Academic Press, London, 1977.

[12] T. Kowalczyk, R.R. Gajewski. An object-oriented graphic user interface for finite element applications. In: *Proceedings of the XII Polish Conference on Computer Methods in Mechanics*, 169–170. Military University of Technology, Warsaw, 1995.

[13] P. Leżański, J. Orkisz, P. Przybylski, R. Schaefer. Fundamentals of an open distributed system for CAD purposes. In: *Proceedings of the XII Polish Conference on Computer Methods in Mechanics*, 194–195. Military University of Technology, Warsaw, 1995.

[14] I.M. Smith, D.V. Griffiths. *Programming the Finite Element Method.* John Wiley, Chichester, 1988.

[15] Th. Zimmermann, Y. Dubois-Pellerin, P. Bomme. Object-oriented finite element programming: I. Governing principles. *Computer Methods in Applied Mechanics and Engineering*, **98**: 291–303, 1992.