TOWARDS THE GENERAL USE OF OBJECT-ORIENTED SOFTWARE ENGINEERING IN EVERYDAY FINITE ELEMENT PROGRAMMING

Gray Farias Moita
Henrique Elias Borges
Vinícius Ferreira de Oliveira Campos
Valdemilson Lopes dos Reis
Centro Federal de Educação Tecnológica de Minas Gerais, Departamento de Pesquisa e Pós-Graduação, Av. Amazonas, 7675, Nova Gameleira, 30510-000, Belo Horizonte, MG, Brasil.
E-mail: gray@dppg.cefetmg.br

Abstract

This paper discusses the convenience of the application of object-oriented software engineering (OOSE) concepts and techniques in the development of software for engineering in order to increase the reliability and performance, enhance the reusability and scalability, and reduce the cost of production and maintenance. It is restricted to the software dedicated to analysis of problems solved via the finite element method (FEM), which has many applications in the field of engineering. Nonetheless, the arguments presented here apply equally well, *mutatis mutandis*, to any software for the engineering.

Keywords: Software engineering, Finite elements, Object-oriented programming, UML, Systems modelling.

1. INTRODUCTION

The development of finite element codes has become a common task in civil, mechanical, aeronautical, electrical (among others) engineering fields for the last decades. However, little attention has been given to the use (or, rather, reuse) of some of these results in future developments. The prevalent approach employed could be summarized as “here is the algorithm, just code it”, and to extend the software “pick some functionality and code it, then pick another functionality and add to it”. There is no planning at all for the software to grow or change, as it would inevitably do. Hence, more and more lines of code are being generated everyday and the re-utilisation of earlier works tends to be low, especially due to the lack of a systematic approach to the development and of a general definition of software development guidelines. There has been some attempts to accomplish this kind of continuity but so far the results are far from satisfactory.

The main problem faced by the current finite element codes, especially those developed at universities, is the lack of a methodological approach for the programming. Coupled with the vast quantity of code generated, some aging more than 30 years, this “disorganised” approach can easily lead to large programs where the maintenance is neglected and reuse is almost impossible. In addition, this deficiency introduces many problems when different researchers try to use the same program. Of course, such problems could be alleviated if some of the directives from the modern OOSE are applied.

Another source of trouble is related to the programming languages utilised in FEM software. Traditionally procedural languages, like Fortran, C and Pascal, have been used to implement the method. Also, these FEM software were often developed in many distinct programming languages. The use of C++, for example, to supersede Fortran usually finds great opposition, although the clear gains are generally recognised. This happens mainly due to the large number of Fortran routines that can be found everywhere and the high cost involved in converting then into some object-oriented programming (OOP) language.

For the last 10 years object-oriented programming languages has been employed in development of engineering software. As far as the application with the Finite Element Method (FEM) is concerned, good reviews can be found in McKenna (1997), Marczak (1999) and Bittencourt *et al*. (1999). Marczak puts together some of the recent publications and an interesting overview of different ways and interpretations of modelling finite element using OOP.

Despite the fact that in the last decade the developers of FEM software started using OOP languages and worrying about reusability and scalability of their software, there is a long way to go through, in order to achieve "industrial strength" FEM software. Fortunately, this situation is gradually changing, with the help from the concepts of object-orientation, as Marczak's work has shown. To be successful in this enterprise, one must complete the paradigm shift initiated in the early '90s, and move towards the application of object-oriented software engineering (OOSE) concepts and techniques in the development of software for engineering.

In this paper the Unified Modelling Language (UML) is used in conjunction with finite elements in order to develop a simple axisymmetric finite element and initiate a validation of this new combination. Some diagrams, associations and notation are shown and its use in the current context is exploited. It must be emphasised here that this work brings results from initial experiments with the *finite element/UML* association and only some early conclusions are drawn and discussed. In this study, the FEM is treated in a pure academic way. This means that the development shown here is used merely to demonstrate the application of the UML for a finite element implementation.

2. HOW CAN OOSE IMPROVE THE MODELLING OF ENGINEERING SOFTWARE

To answer the above question some aspects must be pointed out. Software engineering encompasses a proven (not in the scientific sense, but rather in the empirical sense of "best practices") body of knowledge, tools, techniques and methods to develop computer software products.

There are many ways to develop a software. The two most commons are the procedural approach (also called, algorithmic approach), and the object-oriented approach. In the former, the main building block of a software is the procedure, function or routine, and the main goal of the developer is to decompose a large algorithm into smaller pieces of code (procedures and routines). In the object-oriented approach the main building block of a software is the object, or better, a class of objects. In a phrase, the object is a thing (like a steel bar or a node in a finite element analysis) that has identity (can be distinguished, in some way, from other objects), state (is created; evolves in time, passing through many states; and, eventually, is discarded, i.e., has a lifecycle), and behaviour (can do things in benefit of other objects).

There is nothing inherently wrong with the procedural approach. It can be (and has been) used to develop very good software. On the other hand, there are plenty of software said to be object-oriented (as if OO would be a stamp ensuring quality) that hardly could be called "software". Object-orientation is not a programming language, rather it is a concept (or a

development paradigm) we could use to develop better software systems. Both approaches can benefit a lot from the software engineering techniques.

The main objective of software engineering, both procedural or object-oriented, is to improve software quality by maximising the overall performance, reliability and life cycle of the product, reducing production cost, and minimising the development time, the need for maintenance and the occurrence of errors.

In spite of the increasing publicity of the recent years, software engineering can still be regarded a new branch of the computer science, at least for the non-computer scientists. This often leads to bad development procedures especially due to a lack of experience from the practitioners and of well-disseminated practices necessary to the accomplishment of any product development. However, much effort has been devoted to lead software engineering to the maturity one would expect to find in some of the more traditional fields of engineering.

It is far beyond the scope of this paper to present the principles of the object-oriented software engineering, or to discuss the differences between procedural and object-oriented software engineering (for such, see the excellent book of Pressman, 1997). Instead, our main purpose is to advocate the cause that OOSE can effectively contribute to the development of software for engineering and point out three ideas, which can be easily applied, to improve the quality and reusability of the FEM software. These are the ideas concerning visual modelling and Unified Modelling Language (UML), architectural design, design and architectural patterns.

The basic idea behind the visual models is the abstract representation of the software, encompassing details so that all the complexity can be more easily dealt with, mainly with complex systems where the comprehension is generally very difficult. Instead of starting straight with lines and lines of code, the software engineer has got an alternative and attractive approach to analyse the software (or rather, a general overview of it) without having to worry about specific points and implementation details at the early stages of development.

Visual modelling is a friendly way of representing a problem by mimicking the real-world processes or, better, simulating the environment delineated by the problem. The main advantage of using a visual modelling is that it facilitates the understanding of the requirements of the software to be developed. Therefore, better design and maintenance can be achieved. Well-produced models are a very useful manner to format a clear and easily understandable project. Besides, they can show different views and scenarios within a given development.

In the late ‘80s and early ‘90s several methodologies, and CASE (Computer Aided Software Engineering) tools, have been developed in an attempt to create a standard in visual modelling. Three of them are worthy mentioning because they were created by three top methodologists: Object Modelling Technique (OMT) from James Rumbaugh, Object-Oriented Software Engineering (OOSE) from Ivar Jacobson and Booch Method from Grady Booch. Each modelling approach has its own characteristics and can be used in any given development. However, as always, they also have their drawbacks and that is the main reason to devise a new and standard modelling method. The effort for the unification of the three above methods began by the end of 1994. Soon, the stakeholders from the software industry joined the effort, and finally by the end of 1997 the Unified Modelling Language (UML) was formally accepted as standard by the Object Management Group (OMG). The OMG is an international non-profitable organisation, established in 1989, to promote the development of the theory and practice of the object technology.

Nowadays, the UML is one of the main visual modelling languages there exists. Note that UML is not a programming language, instead, it is a language specially constructed for the purpose of visual modelling. With the help of UML one can obtain a clearer view of what one is trying to develop, one can capture the user’s points of view and, also, easily check the

functionality of the software. If used correctly, UML can eliminate miscommunication due to different modelling terminology, hence, increasing efficiency.

The UML can handle different levels of complexity or different views, so that the system can be displayed in a number of ways and visualised from different standpoints. Reuse of components can be made in a natural manner and modules can be much comfortably manipulated and comprehended. The UML can be used to visualise, specify, construct and document the software to be built and is a handy approach to be used with the traditional software engineering.

Although very young, there is a lot of literature available about UML, as can be seen, for example, in Booch *et al.* (1999), Fowler and Scott (1997), and of course, in the Internet. The main problem we found concerning the literature is that sometimes it becomes out-of-phase with the current standard release of the UML, since OMG is working very hard and fast to keep UML up to date.

If one intends to develop a robust software to be reused by oneself and/or others, and to be changed later (as it certainly will), one should plan for it. So, our first concern should be: what this system will do, how it will be organised, how it will be partitioned into components, how the components would interact, how the components will be allocated for processing in the computers in our network, how can one component be added to the system to increase functionality, how the performance will be affected, etc. These are some of the issues addressed in the architectural design of a software system.

Good architectures are built over well-defined abstraction layers. Each layer represents one coherent abstraction, with a well-defined and controlled interface. Each layer uses the services (functionality) of the lower layers, through their interfaces. Lower layer presents a lower level of abstraction. There is a clear distinction between the interface of a layer and the implementation of the layer itself, such that changes in the implementation of the layer do not affect the layers above (Rechtin, 1991).

Yet, it is virtually impossible to capture such a wide range of issues and demands, in a single picture. This means that the software architect should analyse the software from several different perspectives or viewpoints. How many viewpoints are necessary to ensure the software to be built is fully understood? In a classical paper, Kruchten (1995) argues that one needs “4+1” views of the architecture, which are:

1. *Logical View:* primarily describes the functional requirements of the system, i.e., “what” the system is supposed to provide in terms of services for its end users;
2. *Process View:* this view addresses some non-functional requirements such as performance and system availability. It also addresses issues like fault-tolerance, system integrity, concurrency and distribution;
3. *Development View:* describes the software static organisation in its development environment, i.e., defines how the components of a software would be grouped. A component is a unit of source code that will be used as a building block for the structure of the system;
4. *Physical View:* maps the processes, tasks, objects and every element identified in every other view onto the various processing nodes (like computers and the alike), taking into account the system’s non-functional requirements mentioned above;
5. *Use Case View:* this is a redundant view with the other ones, hence the “+1”. This view drives the process of discovering architectural elements in the other four views. Also, it validates and illustrates the architectural design.

The above architecture has been widely accepted in the community of software engineers. However, it seems that the architectures designed in the last years for the FEM software (Marzak, 1999; McKenna, 1997) do not satisfy the architectural model proposed by Kruchten. Therefore, it is not a surprise when they present problems concerning reuse of parts of codes

(e.g., when someone else wants to use the software to solve a different problem using FEM), they could not be changed easily (e.g., in order to take into account an enhancement of an algorithm), and so on.

Another useful idea that increases the software productivity and quality, is reuse of design patterns. A design pattern describes a common way of modelling (designing) something. In this sense, it is like a template or an example model. However, a design pattern is much more than an example model, it is a solution, or a set of solutions, to a specific designing problem. Most of the design problems faced by the developers are recurring, so are the solutions. This is the main reason why designing patterns are so important ,i.e. , they take the idea of reuse one step further.

In a design pattern, the problem is posed and made clear, then a set of solutions to model (and solve) the problem are presented and explained. Also, some analyses are made concerning the pros and cons of each solution and in which circumstances it works or not. It is also worthy mentioning that design patterns are independent of programming language, so they can be coded in the object-oriented programming language preferred by the user.

Design patterns is still a new research field, but is growing at an astonishing rate. The most influential reference is Gamma *et al.* (1995); another useful reference is Larman (1998). In the Internet there are dozens of web sites dedicated to patterns. The search could begin at the Patterns Home Page (http://hillside.net/patterns/patterns.html) and at the Ward Cunningham's Portland Patterns Repository Page (http://c2.com/ppr/index.html).

If design patterns takes the idea of reuse one step further, then architectural patterns takes it even farther. The main idea behind architectural patterns remains being reutilisation, this time at a highest level of abstraction. Architectural patterns are an even newer research field, also growing very fast. Nowadays, there are some architectures already available in books like Buschmann (1996) and in the Internet. Although they are not completely suitable for software developed for engineering use, they still can be employed as a starting point to construct our own architecture, tailored to fit for our own needs.

3. WHY SHOULD THE UML BE USED

The UML is a formal standard established by the Object Management Group (OMG) in November 1997. Since then, the UML is rapidly becoming, not only the "*de jus*" standard, but also the "*de facto*", since it has been supported and adopted by some of the main industry leaders in the software development arena, including IBM, Microsoft, Hewlett-Packard, Unisys, I-Logix, Oracle, Rational, Texas Instruments, MCI Systemhouse, Intelligcorp, ICON Computing, Ericsson, Andersen Consulting, Sterling Software, and many others. To maintain the UML up to date with the most recent advances of the OOSE and demands from the software developers, the OMG has put together the Revision Task Force which has recently released UML version 1.3, whose adoption voting process in under way (see the OMG web site at http://www.omg.org).

As pointed out by Booch *et al.* (1999), the UML is much more than a bunch of graphical symbols, it is a powerful language that has been built, from the very beginning, to achieve three goals: enable the modelling of systems, from the conception to execution, within the object-oriented paradigm; address the issues of scale, typical in the complex, mission-critical systems; be a modelling language usable by both humans and computers.

The UML defines an expressive and coherent notation fully consistent with the concepts of object-orientation. Some of the characteristics of the UML are:

1. It provides end users, developers, designers, software engineers and software architects with a common standardised language. Hence, contributing to enhance the dialog among these actors;

2. It supports Kruchten's "4+1" architectural views of a system as it evolves throughout the software development lifecycle;
3. It is the only language a developer will need in order to specify, visualise, construct, and document a software system. Also, it is a handy approach to be used with the traditional software engineering techniques;
4. It is a visual modelling language, meaning that it employs a graphical notation;
5. Its vocabulary and rules (semantic and syntax) are precise, unambiguous and complete, such that, everyone can understand;
6. It focuses on the conceptual and physical representation of a system. Thus, making the process of software specification easier, clearer and unambiguous;
7. As a language, it is independent from the chosen software development process. Nonetheless, one must choose any development process in order to model a given software;
8. In conjunction with a well-defined software development process, it contributes to make the reuse of pieces of code easier, and even the reuse of design and architectural patterns;
9. The models built with UML have excellent stability in relation with changes in the specification, i.e., small changes in the software requirements do not imply massive changes in the models;
10. There are several CASE tools of excellent quality available in the market (some of them enabling code generation directly from the model) that supports the UML;
11. The UML has been used successfully in the development of very large and complex software, from aircraft simulation systems to strategic enterprise information systems. It has been also used in the modelling of mission-critical real-time systems;
12. The development of UML has been a collective effort, the contributions coming from the most prominent scientists and software engineers.

4. SMALL FINITE ELEMENT APPLICATION

In this section a simple axisymmetric finite element is used to demonstrate the utilisation of the UML for a finite element implementation. Once again, it should be emphasised that this application has been simply devised to illustrate a combination FEM/UML without much concern with the originality and complexity of the problem. Note that, due to the lack of space, only three diagrams are shown in order to depict the functionality of the visual language employed in the current context.

In the development to be described below, each diagram is briefly explained according to the usual UML terminology.

Use Case Diagram:

The use cases describe the functional requirements of the system as observed by the external actors. The actor could be viewed as something that interacts with the system and can be a user, a device or another system. Figure 1 presents the Use Case diagram for the present case study.

In the current use case model, only one actor has been created, namely, the user running the software. The diagram also shows the use cases: check data, define finite element, assemble matrix, assemble vector and solve system. They define the ways the system functions. The use case diagram helps to understand how the system works and the possible interactions and gives a general outlook of the development of the project. Note that in this case, only actor interacts with the program but this is not always the case (Booch *et al.*, 1999).

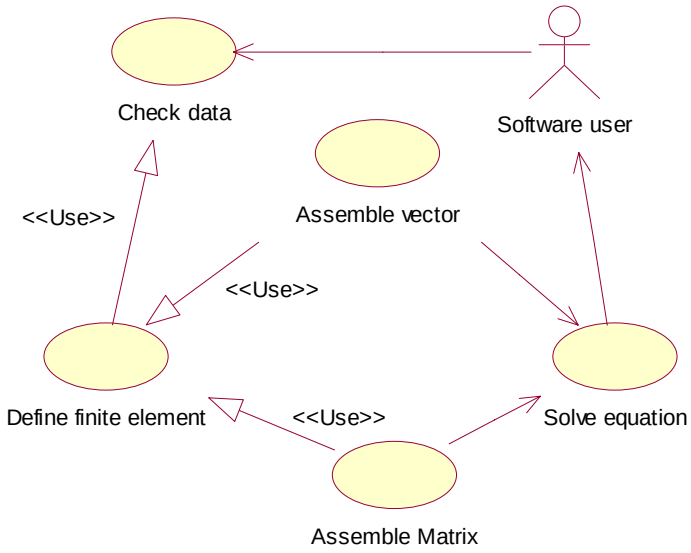

Figure 1. Use Case diagram for the finite element case study

Class Diagram

A class diagram represents the static structure of a given system. Classes are templates to create objects with common characteristics, i.e., pertaining to a certain category. Within the class diagram five main relationship can be used, namely, specialisation, association, aggregation, composition and dependency.

There are several class diagrams for this current development. With the purpose of illustrating this case study, only one of them is shown in Figure 2.

Due to the characteristic of the problem, the several diagrams are grouped together into packages that must represent some kind of unity or module within the system. The class diagram of Figure 2 displays the classes in the package vector, the relationship among the classes as well as the kind of relationship (association or aggregation) and the cardinality.

As can be seen in the diagram, nodalDisplacementVector, loadVector and boundaryConditionVector are specialised classes (or subclasses), derived from the abstract class vector. The latter only exists to encompass the common attributes and methods to be use by the child classes. Also, the method checkConsistency is inherited by the three subclasses although each one implements its own code for the particular consistency test needed. This is an example of the so-called polymorphism.

The main class diagram, encompassing the different packages of classes, for a given case study can be devised. In the current system data, the following packages can be thought: data, vectors and matrices and solution algorithms. Each should present high internal cohesion and low coupling with other packages.

Sequence Diagram

The sequence diagram depicts a time-wise event flow for each use case. It is a two-dimensional diagram with time represented vertically and objects (not classes) in the horizontal position, showing the sequence of messages sent amongst the objects. The diagram of Figure 3 displays the time sequence of events of the use case Solve equation.

Each the sequence diagram is based on the event flow created for a specific use case, e.g., the event flow for the use case shown in Figure 3 is:

Preconditions:

The use case matrix assembly and vector assembly must have been successfully completed to allow for this use case to initiate.

Main Flow:

1. *Receives the global stiffness matrix*
2. *Receives the global load vector*
3. *Picks up the defined solution algorithm*
4. *Solves the finite element equation to determine the displacements*
   K a = f, where K = global stiffness matrix; a = displacements vector; f = load vector
5. *Sends displacements to the object myDisplacementVector of the class displacementVector.*
6. *Displays the results (the displacements) to the object "anybody" of the stereotyped actor class named user.*

Alternative Flow:

If an error occur during the solving process due to wrong data, the program sends the users an error message and finishes the execution.

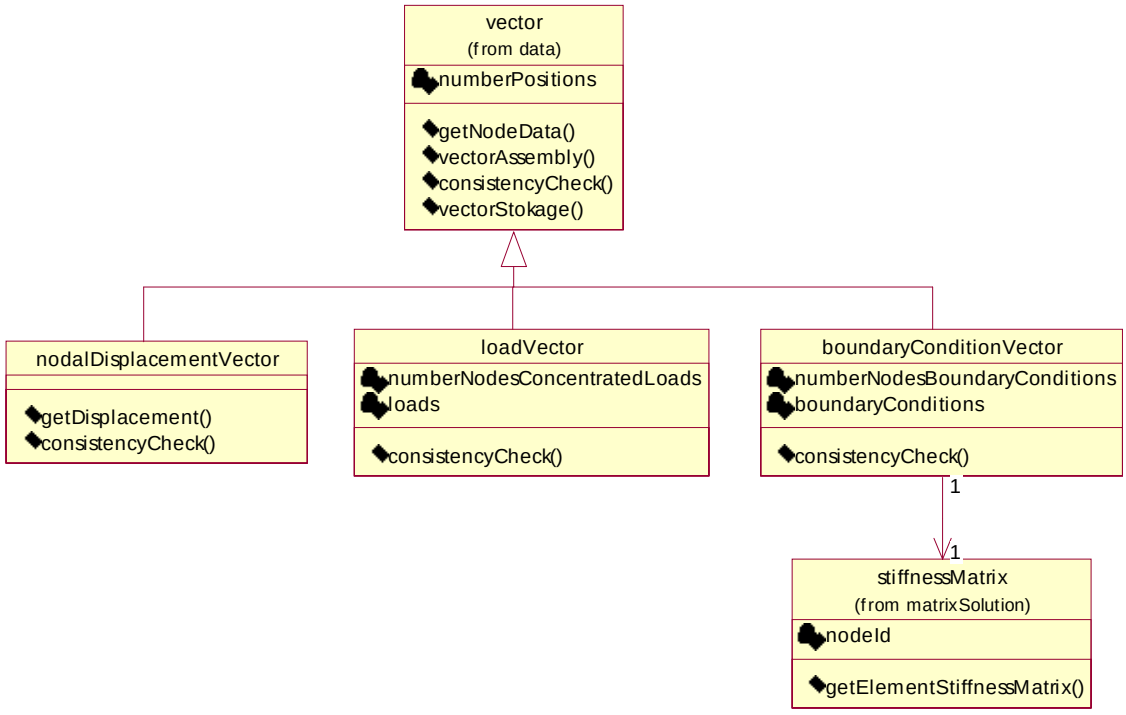

Figure 2. Class diagram for the package vector

5. FINAL REMARKS

The paper deals with the use of object-oriented software engineering for finite element programming in conjunction with the introduction of visual modelling techniques. The suitability of the use of the UML is exploited and a general explanation on the subject is given. The main points discussed are reusability, scalability, comprehensibility, reduction of production and maintenance costs as well as prevention of errors.

The main objective was to shed some light on the subject, mainly regarded with the association of finite elements and the new software engineering tendencies, in order to motivate the finite element community to adopt these modern ideas.

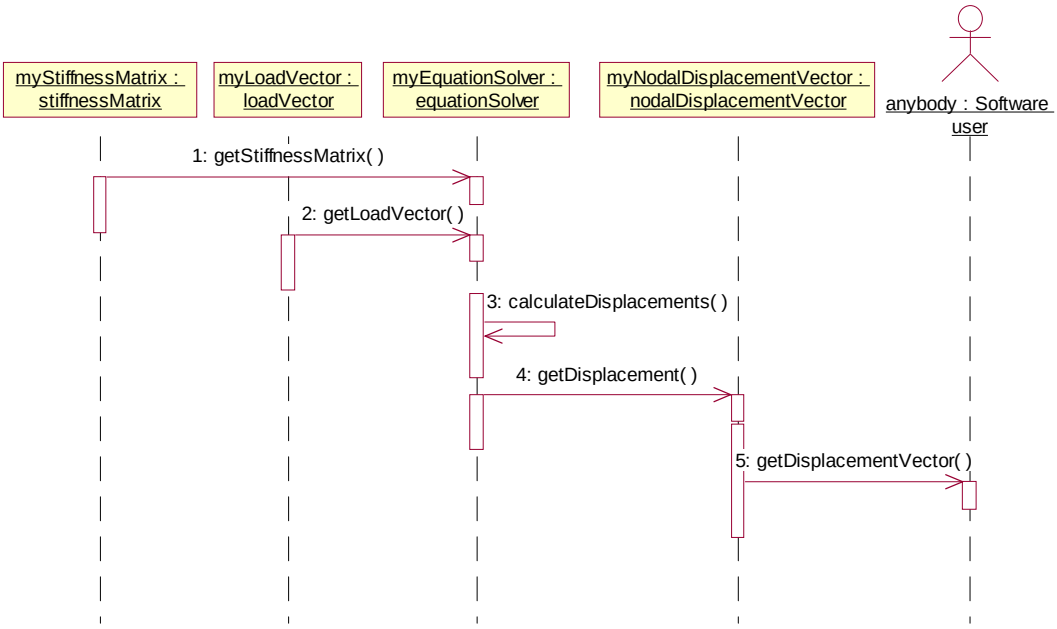

Figure 3. Sequence diagram for the use case Solve equation

6. ACKNOWLEDGEMENT

The authors would like to express their gratitude to CNPq, FAPEMIG and FINEP for their financial support.

7. REFERENCES

- Bittencourt, M.L., Guimarães, A.S. and Feijóo, R.A., 1999, "Elementos Finitos Orientador por Objetos", Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería, Vol. 15 (3), pp. 343-355.
- Booch, G., Rumbaugh, J. and Jacobson, I., 1999, "The Unified Modelling Language User Guide", Addison-Wesley, Object Technology Series, 4th Printing.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., 1996, "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons.
- Fowler, M. and Scott, K., 1997, "UML Distilled: Applying the Standard Object Modelling Language", Addison-Wesley, Object Technology Series, 11th Printing.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [Gang of Four], 1995, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley.
- Kruchten, P.B., 1995, "The 4+1 View Model of Architecture", IEEE Software, pp. 43-50.
- Larman, C., 1998, "Applying UML and Patterns", Prentice Hall.
- Marczak, R. J., 1999, "Uma Revisão Parcial de Arquiteturas Orientadas a Objetos Para Programas de Elementos Finitos", Proceedings of the 15th Brazilian Congress of Mechanical Engineering, CD-Rom, Águas de Lindóia, SP, Brazil.
- McKenna, F.T., 1997, "Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing", PhD thesis, University of California, Berkeley.

- Pressman, R.S., 1997, “Software Engineering: A Practitioner’s Approach”, McGraw-Hill, 4th Edition.
- Rechtin, E., 1991, “Systems Architecting: Creating and Building Complex Systems”, Prentice-Hall.