A FORMAL UML-RELIANT SOFTWARE ENGINEERING APPROACH TO FINITE ELEMENT SOFTWARE DEVELOPMENT FOR ELECTROMAGNETIC FIELD PROBLEMS

SAMUEL RATNAJEEVAN HERBERT HOOLE, THIRUCHELVAM ARUDCHELVAM

Key words: Electromagnetic field, Finite element, UML.

Much of today’s finite elements-based software for electromagnetic field computation has been developed on an ad hoc basis, often as an outgrowth of ongoing research whether at universities or agencies like NASA. “Design” is seen as the execution of computational algorithms yielding the computed design, rather than the holistic process that begins with the design of the software itself and ends in the computations using that software. As a result finite element software product development has not been informed by the benefits of starting with a rigorous requirements analysis and going through the normally mandated design process in any rational software engineering development process. This paper examines such a development and identifies the various benefits that arise from that process. Of particular note is the ability to have a list of software components from which we pick the most appropriate to the problem being addressed – thereby giving users choice to suit their particular computational environment, be it in methods or hardware; and the ability to implement a model of the design using UML and then transform that into source code using modern facilities. The latter makes porting to new programming languages easy and ensures that the finite element code is proper rather than simply working. Reengineering legacy software by transforming existing code to UML for analysis becomes possible. And the planning document leads to all goals being realised and is easily transformed into a user manual.

1. THE SOFTWARE ENGINEERING LIFECYCLE

A formal software development process is carefully defined in terms of the software engineering lifecycle for the best design and product outcome. The development of finite element field computation software, however, commenced in an era when the discipline of software engineering had not even begun, let alone matured. Indeed much of the development was in languages such as FORTRAN that today are not considered the best. The focus was on the computation rather than on a continuous process from software development to computation using the

* Department of Engineering and Science, Rensselaer Polytechnic Institute, 275 Windsor Street, Hartford, CT 06120, USA, E-mail: HooleR@RPI.Edu; Fax: +1-860-548-5319

software developed. The legacy portfolios so developed had much investment value. This prevents rewriting, exploiting the modern tools available today.

As shown in Figs. 1 and 2 (adapted from [1]), a software engineering lifecycle begins with requirements elicitation from the client to requirements specification and analysis, followed by system design (decomposing the system and addressing design goals) and object design (identifying reuse patterns – of importance in using legacy code – and interfaces) and finally mapping objects to code and testing and maintenance – with natural provisions to revisit stages.

Fig. 1 describes the problem statement in finite element code development. A document is cooperatively written by the client and project management that briefly describes the scope of the system, including high-level requirements, target environment, deliverables, and acceptance criteria. In requirement elicitation the client describes to the developer the system to be developed. The client, the developers and the users work together, contributing their perspectives. Nearly always it is the user who defines what he wants to the developer. The developer in turn guides the user who may not know what is possible. In rare instances, as when one of the authors developed code for a venture capital firm, he had to define these [2]. Such stage is requirements specification and serves as a contract between the client and the developers. It is structured/formalised during analysis to produce an analysis model. Both requirement specification and analysis yield the same information, but differ in the language and the notation; the requirements are set in
natural language, whereas the analysis model is usually expressed in formal or semi-formal notation. Requirements specification supports the communication between client and users; the analysis among developers [1].

Functional requirements describe the interactions between the system and its environment independently of its implementation. The environment includes the user and any other external system with which the system interacts. The functional model, in UML (a modelling-not programming-language) with use case diagrams, describes the users’ perspective the functionality of the system. Non-functional requirements describe user-visible aspects of the system that are unrelated to functionality: a) implementation requirements (programming language, hardware platforms, etc.) and (b) interface, operations, packaging and legal requirements.

Fig. 3 – Collaboration Diagram for FEM.

The dynamic model (represented in UML with interaction-, statechart-, and activity-diagrams) describes the internal behaviour of the system. Interaction diagrams (e.g. the collaboration diagram of Fig. 3) describe behaviour as a sequence of messages exchanged between a set of objects (the user and hardware as well as methods). Statechart diagrams describe behaviour in terms of the states of an individual object and its possible transitions. The object model is represented as UML class diagrams as in Fig. 4. It describes the structure of the system in terms of objects, including attributes, associations and operations. During requirement analysis, the object model starts as the analysis object model and describes the relevant application concepts. During system design, the object model is refined into the system design object model and includes descriptions of the subsystem interfaces. During object design, the object model is refined into the object design model and includes detailed descriptions of solution objects as in Fig. 2.
Developers define the system design model, including the design goals of the project, and decompose the system into smaller subsystems that can be realised by individual teams. Most, perhaps all, finite element code development has begun here, such development having been task oriented with specific results in mind. The stages from here onwards have been followed, but cannot represent “best practice” without the preceding design stage. System design also leads to the selection of the hardware/software platform, present data management strategy, global control flow, access control policy, and boundary condition strategies. For example, we need to examine the boundary conditions of the system — that is, to decide how the system is started, initialised, and shutdown — and we need to define how we deal with major failures such as data corruption and network outages, whether they are caused by software error or power outage.

During object design, developers define custom objects to bridge the gap between the analysis model and the hardware/software platform. This includes specifying object and subsystem interfaces, selecting off-the-shelf components, restructuring the object model to attain design goals and optimizing the object model for performance. Object design results in the object design model.

When the design part is completed, the model should be mapped to code. This is called implementation/coding. Fig. 5 shows the concept of transformation from model to code [1]. A transformation that is applied to an object model and results in another object model is called model transformation [1]. The purpose of object model transformation is to simplify or optimize the original model, bringing it closer to a system that complies with all requirements in the specification. Forward engineering is applied to a set of model elements and results in a set of corresponding source code statements, such as a class declaration, a Java
expression, or a database schema. The purpose of forward engineering is to maintain a strong correspondence between the object design model and the code, and to reduce the number of errors introduced during implementation, thereby decreasing implementation effort.

Reverse engineering is applied to a set of source code elements and results in a set of model elements. The purpose of this type of transformation is to recreate the model for the existing system, either because the model was lost or never created, or because it became out of correspondence with the source code. Reverse engineering creates a UML class for each class declaration statement, adds an attribute for each, and adds an operation for each method. This reveals the thinking behind the design of the code and facilitates its improvement. Refactoring is a transformation of a source code that improves its readability or modifiability without changing system behaviour. It is to improve the design of a working system examining a specific field or method of a class.

Testing is an activity needs no detailed description here It includes unit testing, integration testing, system testing and usability testing. Maintenance is concerned with the resolution of software errors, faults, and failures after the delivery of the system.

2. THE DISADVANTAGES OF AD HOC APPROACHES

An elaborate description of the software engineering process has been given to emphasize what is lost in ad hoc code development. Often as code is developed in a research laboratory, the goal is results, not user convenience. The hardware
platforms are what is available. Variables will be open to the entire program or to parts without proper planning. The program stands so long it runs. Thus when reengineering code becomes necessary and the code is used to map back to the corresponding classes, errors will arise. When code is developed *ad hoc*, the development begins directly at coding, the penultimate stage of the software engineering lifecycle. This paper attempts to demonstrate the proper approach to finite element code development and the advantages that accrue thereby.

3. THE ADVANTAGES OF A FORMAL PROCESS

The advantages of the formal approach to finite element code development are: a) All requirements being met. When we go through the requirements analysis process, we would not need to worry about say having to plot equipotential lines from quadrilateral elements after developing routines for a triangular mesh. As we have developed code [3], a general polygonal element is assumed and the same methods can be used by passing the number of sides of the polygonal element as a parameter. Reengineering as an afterthought is very costly. b) An efficient design. For example, as illustrated below under components, many matrix operations like matrix $[A]$ times vector $\{x\}$, $\{x\}'[A]\{x\}$ and decomposition of a matrix $[A]$ into its Cholesky factors are common to many operations. By defining these classes and operations with these classes, much efficiency is realized in code development and debugging c) The ability to generate code directly and automatically [1] from the UML-description of the design [4]. Today there is software available that can convert a UML design directly to code. These features can and must be exploited.

The requirements are gathered from finite element method experts before the design. After formalizing the requirements properly, they are verified and clarified with the experts. Their additional suggestions are also included. What results in the requirements document is a general design (not for a specific application). The components can be taken separately and designed into another package for another purpose. Therefore it is said to be a multipurpose design. Further, it is accepted by the finite element method experts prior to implementation. As a result of this approach, software can be seen as components that are pressed into service by gluing together “prefabricated” parts [5]. This concept neatly fits in with the idea of pattern reuse in software engineering and has been developed to some extent in [5–7] where independently generated software is reused employing a standardized data format permitting free exchange.

4. UML-DESCRIPTION OF MATRIX SOLVERS

This design approach concept can be applied to all finite element applications. In this particular section, matrix solution is taken just for the purpose
of explaining the concept. Fig. 6 shows the UML use case diagram of resource allocation in adopting this design approach for finite element code development. This was arrived at using academic experts and engineers in industry in MS classes who work with field computation as our “clients” – we being the software project team – eliciting their requirements for matrix solvers in the course of their work. Other requirements in mesh generation, post-processing, interfacing with other packages [5-7] were also elicited but are not detailed here. An actor is an external entity that needs to exchange information with the system. An actor can represent either a user role or be another system. In the above diagram “Project Manager” is a user who will interact with the system to choose a matrix solver. For a classic single processor PC as actor, the ICCG solver would be appropriate and for a distributed system the old fashioned Gauss with message passing would be the choice. Gauss_Seidel is an extended version of Gauss iterations for the exceptional situation of nonsymmetric matrices as with a Galerkin formulation of the Poisson equation without recourse to Green’s identity [3]. Code is developed with all methods. The user may have different hardware systems and the option is given to choose any method he thinks more suitable for his computer resources as actors.

Working with experts, we come up with the design of classes in Fig. 4 which in turn would lead to objects. Fig. 4 gives an outline of the MatrixSolver which uses the matrix class which in turn uses the vector class. The detailed design with properties and methods is not shown here. But this gives the overall idea of the design. The class MatrixSolver contains different types of solvers which are defined as subclasses. Each subclass uses different methods suitable for different types of matrices. For some types of matrices, obviously, more than one solver exists; for example direct inversion or Gauss triangulation, can be used to solve a general matrix as may be required in curve fitting for magnetic material properties. Matrices are represented in different ways depending on the data they consist of. For example, the matrix from the finite element formulation is sparse. Accordingly sparse or profile data structures are employed [3]. Fig. 4 also shows the different types of matrices we may deal with. In the design of the matrix class, these are analysed. Depending on the category of the matrix, the solver type is selected automatically using the facility of polymorphism in Java. Object oriented concepts, especially polymorphism, play an important role in this design. Further, the components of this particular design can be utilised for some other software designs and for different applications as well. That is a marked advantage of this design approach. This is where we implement the software engineering concept of “Pattern Reuse”. Hence this particular design is considered multipurpose design.
In this design, the class “vector” has subclasses such as “integerVector”, “rowVector” and “columnVector”. The nomenclature is self-explanatory. Vector is used to store elements in an array. “integerVector” is used to store integers and is used as a data structure used in representing matrices. “rowVector” and “columnVector” are used to store arrays of decimal numbers (double in Java). Even though, as a programming concept, the structures of “rowVector” and “columnVector” are the same, here in this particular design, as matrix concepts, they were considered different. By having the class “vector” and its subclasses, it is very easy for us to design the matrices and to define some matrix operations. For example, when a sparse symmetric matrix is represented, “rowVector” is used to store only the nonzero elements, while the locations of diagonal elements of rows and column numbers of nonzero elements are stored in “integerVector” separately [3]. With this representation, the sparse symmetric matrix is used efficiently in terms of memory and CPU time.

5. ADVANTAGE – COMPONENTS

The concept of components gives users choice in piecing together the methods and processes they prefer rather than imposing that choice through code with pre-chosen methods. Fig. 3 shows the components-based finite element solution process. The user gives the essential problem data. Finite element components are available for different element shapes, orders, material characteristics, matrix methods, etc. Once the user chooses the finite element
components, the system communicates with the databank to gather the needed information and methods. The matrix equation is solved by the chosen solver.

Fig. 7 shows the class diagram for MatrixSolver which contains methods to compare the solutions, solution time of different solvers, etc. This leads to ease of facility in checking the efficiency of different solvers for different applications. By having these facilities, a system can be developed to give the best solution method for a given industrial magnetic field problems. Likewise, all the other components, i.e. classes, are specialised for developing an efficient system.

Fig. 8 shows the automatically generated Java code from our class designs using ArgoUML™ [9] for the class MatrixSolver shown in Fig. 7. Reverse engineering can now be invoked to generate Java code. Several other UML modelling tools are also now available with these facilities, although we have employed ArgoUML. After creating Java files like this, we have to fill out the code for the operations to be executed in the skeletal form of Fig. 8. That is all we have to do. This cuts down our work as to what is passed, what is visible where, etc..

6. ADVANTAGE – INTERFACE, SEQUENCE

Once the analysis and design are completed with carefully defined interfaces, the UML diagrams would yield the Object Design Document (ODD) leading automatically to the basic skeletal core of the source code as in Fig. 8. At present, several open source UML modelling tools are available to convert from UML to a specific Object Oriented Language. For example, ArgoUML, [9] can be used to get the outline of the source code in either C++ or Java. Since the available design can be converted into different programming languages, this process is flexible for us engineers because we can use our own choice of programming language.

Collaboration diagrams impose a sequence of operations, containing the same information as sequence diagrams but in a different way [1]. From the collaboration diagram of Fig. 3, it may be noted that it is after preparing finite element data that the matrix equation is assembled and after equation assembly that the matrix solver is activated; and indeed, after this that postprocessing is invoked. This sequence is inviolable. Yet, in many finite element programs, one may violate this sequence because it is assumed that the user knows what to do. But in this formal approach to code development that chance is obviated.

In an application, sometimes, some properties may need to be protected from the outside world. This is a common issue. In this design approach, it is achieved by using the access controllers such as protected and private in class definitions.
7. CONCLUSIONS

Developments in software engineering over the past decade permit the design process for electromagnetic products to include even the design of the software that does our computations. This paper studies and identifies the advantages of undertaking finite element code development using this approach. This is best practice. UML communicates the design model; the programming language then communicates the solution [3] as a holistic unification of the electromagnetic design process. The advantages are components-based design, automatic code generation, proper interfaces between methods and a ready-made user manual in the initial specification document which can also be used to verify deliverables. This approach opens up reverse engineering where existing code can be transformed into UML to understand the thinking that went into legacy systems and thereby to test legacy designs and improve them [1, 10].

ACKNOWLEDGEMENTS

T. Arudchelvam thanks his employer, Wayamba University of Sri Lanka, for doctoral study leave; University of Peradeniya Sri Lanka where he is a doctoral student; and RPI for facilities to work on his thesis as a Visiting Fellow.

Received on July 27, 2010

REFERENCES