# Reverse Engineering as a Means of Improving and Adapting Legacy Finite Element Code

S. Ratnajeevan H. Hoole* and T. Arudchelvam
*Department of Engineering and Science, Rensselaer Polytechnic Institute*
275 Windsor Street, Hartford, CT 06120, USA
e-mail: HooleR@rpi.edu, arudct@rpi.edu

***Abstract*—The development of code for finite elements-based field computation has been going on at a pace since the 1970s, yielding code that was not put through the software lifecycle – where code is developed through a sequential process of requirements elicitation from the user/client to design, analysis, implementation and testing (with loops going back from the second stage onwards as dissatisfactions are identified or questions arise) and release and maintenance. As a result, today we have legacy code running into millions of lines, implemented without planning and not using proper state-of-the-art software design tools. It is necessary to redo this code to exploit object oriented facilities and make corrections or run on the web with Java. Object oriented code's principal advantage is reusability. It is ideal for describing autonomous agents so that values inside a method are private unless otherwise so provided – that is encapsulation makes programming neat and less error-prone in unexpected situations. Recent advances in software make such reverse engineering/reengineering of this code into object oriented form possible. The purpose of this paper is to show how existing finite element code can be reverse/re-engineered to improve it. Taking sections of working finite element code, especially matrix computation for equation solution as examples, we put it through reverse engineering to arrive at the effective UML design by which development was done and then translate it to Java. This then is the starting point for analyzing the design and improving it without having to throw away any of the old code.**

***Index Terms*— Reverse Engineering, Reengineering, UML, Class and Sequence Diagrams, Java, FORTRAN, Legacy Software, Finite Elements.**

## I. Finite Element Code

SOFTWARE ENGINEERING, has today matured as a discipline [1-3]. In developing code, strict rules are specified as to how. In a multistage, sequential effort (Fig. 1), we start with requirements elicitation from the user/client to design, analysis, implementation and testing (with loops going back from the second stage onwards as dissatisfactions are identified or questions arise) and finally release and maintenance.

Today as work moves into component-based software with user choice in putting together different methods to do the job at hand [4], it is extremely important that components match and have the correct interfaces. This component compatibility and reuse are ensured when software goes through the formal design process [4].

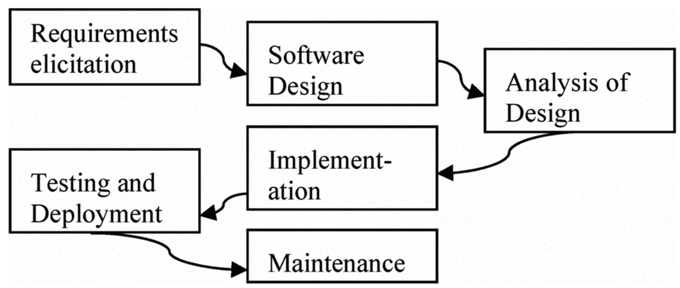



Fig. 1: The Software Engineering Lifecycle

Field computation, whether in electromagnetics or in any of the myriad other sister disciplines with analogous equations, developed at a pace growing exponentially from the late 1960s onwards. Beginning with finite difference network models [5] off which potentials were measured and thereafter moving to real computation on digital computers with finite differences, finite element code has been developed from the 1960s. This was a time when software engineering had not emerged yet as a discipline so code like NASTRAN (NASA's Structural Analysis program [6]) was confined to civil/structural applications and was not publicly available. By the end of the 1970s the first large scale finite element code for electromagnetic field problems had become available. Some of the code from research labs of universities was being marketed by the more entrepreneurial professors and lecturers.

These developments were, correctly speaking, *ad hoc*. A senior academic in his or her research laboratory would develop code through theses. His particular methods as implemented as particular code, would be the mainstay of computation for other research students who follow in the group as it was convenient. As time went on, each group was married to its code and every student coming into that university worked with the same methods as rewriting code for another method would add immensely to the period to tackle a problem. Today several of these codes flowing out of university laboratories are marketed. Some of these have had their maintenance taken over by professional computer scientists. However, because of the vast investments required to change over, particularly to new languages, the code is still for the most part not fully rewritten but rather massaged to work. The rare exceptions were i) when academics handed over their work to professional developers who might have redeveloped code from scratch; ii) when newer research groups came on to the scene having the advantage of choosing

Manuscript received: Aug. 14, 2009. Accepted: Oct. 23, 2009. Revised: Nov. 14, 2009.

S.R.H. Hoole is Professor of Engineering and Science at Rensselaer Polytechnic Institute. ('Phone: 860-540-5356; e-mail: HooleR@rpi.edu). Mr. T. Arudchelvam is Lecturer, Wayamba University of Sri Lanka and a doctoral student at University of Peradeniya working as a Research Fellow at RPI.



*978-1-4244-4837-1/09/$25.00 ©2009 IEEE*

the best of the mathematical methods, the best of the languages and the best of the user interfaces [5, 7]. And yet, there is little evidence that even their programs went through, indeed were put through, the formal software lifecycle; and iii) Corporate entities with resources developed code. Nonetheless, the bulk of code today remains *ad hoc* in nature.

Such *ad hoc* code was never informed by the modern rules of software engineering. Much of the code was in FORTRAN [6] and this code reaching the magnitude of millions of lines could not be practicably redeveloped in more modern languages with their object oriented features. Where new programming languages and user interfaces became available, the older code was called from shells written in languages like C++. There are many examples of NASTRAN, the FORTRAN code, being called by some of the shells written in the new languages with easier GUIs, for example a modern version of NASTRAN by MacNeal-Schwendler Corp [6].

Thus the state-of-the-art today is code implementing mathematically well-researched and powerful methods developed in research labs to solve particular problems and not the most suitable code in terms of well-designed modules passing the rules of software engineering to have sound interfaces and code design when subjected to standard methods of analysis.

This paper examines the issue of reengineering or reverse engineering this legacy code to bring it in to line with the norms of modern-day software engineering principles. Besides such reengineering becomes necessary if we wish to correct or improve code, verify the design of the code or use object oriented languages to adopt best practice and facilitate reuse. Indeed, even when the code is developed in C or C++, it may be desirable to reengineer the code to run in Java so that it may be offered over the Internet [8].

## II. RE-ENGINEERING FINITE ELEMENT CODE

UML diagrams [2] are the main modern software tool for analyzing and designing code, in this instance finite elements code. Class diagrams in UML are used to analyze and design interfaces between modules, a must to facilitate reuse and when components are put together [4]. Reengineering or reverse engineering is the engineering (i.e., re-designing) of existing code that may be erroneous or non-ideal for having been developed in an earlier era – so called legacy code [9-14]. A powerful UML feature to re-engineer code is to give object oriented source code which might be written in Java, C++, etc. as input and generate the class diagrams that capture the fundamental design that went into generating the code [15]. (Conversion of FORTRAN code which is not object oriented is discussed separately in Section IV.) That is, the object oriented code is used to create the presumed class design that went into the code. This generated class diagram then would yield any mistakes in the code. The corrected class diagram can then be used to recreate the code with corrections. This corrected code essentially does not give the detailed algorithms but rather the object oriented code with the headers and passing parameters for all functions [4]. Since the original code was working code, much of the filling in can be done from the original working code.

For example, previously existing Java code of the class called Matrix shown in Fig. 2 is used to create the class

```
public abstract class Matrix
{
    private int columns;
    private int rows;

    public int rowSize()
            { return rows;}
    public int columnSize()
            { return columns; }

    public abstract void display();
    public abstract double elementAt(int i, int j);
    public abstract Matrix multiply(double K);
    public abstract Matrix multiply(Matrix B);
     public abstract columnVector multiply(columnVector B);
    public abstract Matrix add(Matrix B);
    public abstract Matrix sub(Matrix B);
    public abstract void swapRows(int i, int j);
    public abstract void swapColumns(int i, int j);
     public abstract Matrix transpose();
    public abstract Matrix inverse();
}
```

Fig. 2. Java code used to develop class diagram

diagram, shown in Fig. 3, using ArgoUML™ [4]. After creating the class diagram, access level modifiers of the fields "columns" and "rows" were changed from "private" to "protected". In ArgoUML™, facility is given to select a suitable access

```
Matrix
--------------------------------------------
columns: int
rows: int
--------------------------------------------
rowSize(): int
columnSize(): int
display(): void
elementAt(i: int, j:int): double
multiply(k:double): Matrix
multiply(B:Matrix): Matrix
multiply(B:columnVector): columnVector
add(B:Matrix):Matrix
sub(B:Matrix):Matrix
swapRows(i:int, j:int): void
swapColumns(i:int, j:int): void
transpose(): Matrix
inverse(): Matrix
```

Fig. 3. Generated class Matrix

```
public abstract class Matrix
{
    protected int columns;              // **********  changed
    protected int rows;                 // **********  changed

    public int rowSize()
            { return rows;}
    public int columnSize()
            { return columns; }

    public abstract void display();
    public abstract double elementAt(int i, int j);
    public abstract Matrix multiply(double K);
    public abstract Matrix multiply(Matrix B);
     public abstract columnVector multiply(columnVector B);
    public abstract Matrix add(Matrix B);
    public abstract Matrix sub(Matrix B);
    public abstract void swapRows(int i, int j);
    public abstract void swapColumns(int i, int j);
     public abstract Matrix transpose();
    public abstract Matrix inverse();
}
```

Fig. 4. Generated Java code after modifying the class Matrix in Fig. 3.

level modifier using "radio buttons". Then the reverse engineering facility is used to create Java code, shown in Fig. 4, for the modified class diagram. The changes are shown in lines 3 and 4 in Fig. 4. The created class diagram can be modified into any form the developer wants.

Reverse engineering can be used to convert from one language to another. Assume that source code in Java is to be converted to C++. After creating the class diagram using the

```
#ifndef Matrix_h
#define Matrix_h
#include "Matrix.h"
#include "columnVector.h"
class Matrix {
   /* {src_lang=cpp}*/
 public:
  virtual int rowSize();
  virtual int columnSize();
  virtual void display() = 0;
  virtual double elementAt(int i, int j) = 0;
  virtual Matrix &multiply(double K) = 0;
  virtual Matrix &multiply(Matrix &B) = 0;
  virtual columnVector multiply(columnVector B) = 0;
  virtual Matrix &add(Matrix &B) = 0;
  virtual Matrix &sub(Matrix &B) = 0;
  virtual void swapRows(int i, int j) = 0
  virtual void swapColumns(int i, int j) = 0;
  virtual Matrix &transpose() = 0;
  virtual Matrix &inverse() = 0;
 protected:
  int columns;
  int rows;
};
#endif // Matrix_h
```

Fig. 5. Generated C++ code for Matrix.h

existing Java code, reverse engineering is used to create C++ code. Figs. 5 and 6 describe C++ code generated from the class Matrix shown in Fig. 3. The generated code can be edited later to improve efficiency or to look simpler. When the code is generated in C++, header files with extension .h, may also be created (Fig 5). That header file should be included as in line #1 in Fig. 6 in the source file. Some lines tell us not to delete them but they can be deleted when we deal with the

```
#include "Matrix.h"
 {
   /* {src_lang=cpp}*/
int Matrix::rowSize()
         // don't delete the following line as it's needed to preserve
         source code //of this autogenerated element
         // section -64--88-0-101-2b4ec7fa:122a7dccddb:-
         8000:0000000000000EEC begin
{ return rows;}
       // section -64--88-0-101-2b4ec7fa:122a7dccddb:-
       //8000:0000000000000EEC end
       // don't delete the previous line as it's needed to preserve
       source code //of this autogenerated element
int Matrix::columnSize()
       // don't delete the following line as it's needed to preserve
       //source code ///of this autogenerated element
       //section -64--88-0-101-2b4ec7fa:122a7dccddb:-
       //8000:0000000000000EF1 begin
{ return columns;}
       // section -64--88-0-101-2b4ec7fa:122a7dccddb:-
       //8000:0000000000000EF1 end
       // don't delete the previous line as it's needed to preserve
       //source code //of this autogenerated element
```

Fig. 6. Generated C++ code for Matrix.cpp

C++ program. Those lines should not be deleted to preserve the source code, only if we are going to reengineer the C++ code.

An attempt was made to convert a fully object oriented finite element software written in C++ [4] to Java code using

```
/*---------EquationSet.h---------------------*
 *---------07:04:2001------------------------*
 Contains the difinitions of class CEquationSet
*/
#ifndef EQUATIONSETDEF
#define EQUATIONSETDEF
#include "Matrics.h"

class CEquationSet{
public:
     CEquationSet(CMatric* A, CMatric* x, CMatric *b);
 int      Solve();
private:
     CMatric* pA; // A * x = b format
 CMatric*     px;
 CMatric*     pb;
     int N,M;    //Dimensions of pA

     void rDiv(int row,double n);
     void rMulSub(int row1,int row2,double m);        void
rSwap(int row1,int row2);
     int Pivot(int row);
};
#endif
```

Fig. 7. C++ code for class CEquationSet

```
public class CEquationSet {
      private CMatric pA;
      private CMatric px;
      private CMatric pb;
      private int n;
 private      int m;
      public CEquationSet(CMatric aA, CMatric aX, CMatric aB) {
         throw new UnsupportedOperationException();
 }
      public int Solve() {
         throw new UnsupportedOperationException();
 }
      private void rDiv(int aRow, double aN) {
         throw new UnsupportedOperationException();
 }
      private void rMulSub(int aRow1, int aRow2, double aM) {
         throw new UnsupportedOperationException();
 }
      private void rSwap(int aRow1, int aRow2) {
         throw new UnsupportedOperationException();
 }
      private int Pivot(int aRow) {
         throw new UnsupportedOperationException();
 }
}
```

Fig. 8. Generated Java code for class CEquationSet

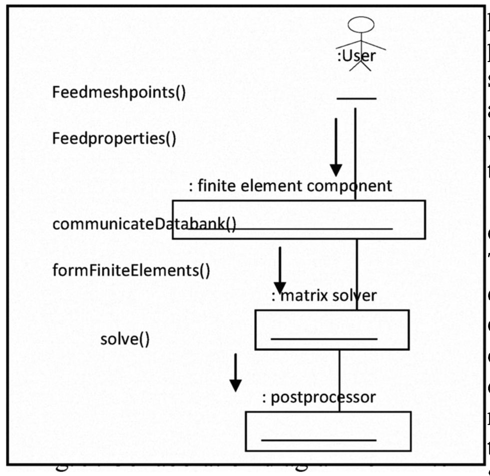
re es. In the Java code the outline of the classes with property names and method names are generated. The body of the Java code needs to be filled in order to get the complete Java code. First the C++ code is reengineered to get the class diagrams and then from the class diagram, the Java code is generated using forward engineering. For example, a class called "CEquationSet" written in C++ shown Fig. 7 is converted to the Java code in Fig. 8. The total translation yields extensive output, but here only a sample is provided.

By converting finite element software written in C++ into Java, it is easy for us to put it on the internet.

Further, when software code is reverse engineered, first, for example, a project report or a manual may need to be read to understand the project properly; problems arise when there are differences between the report and the program. For example, the name of a class used in the project report may be different from the name used in the program. If a programmer reads the report first and then tries the reverse engineering, he may be confused because of the name conflicts. When reverse engineering is done, conflicts are immediately apparent and lend themselves to easy correction; and care must be taken to not introduce further misunderstandings and conflicts like this.

## III. Sequence Diagrams in Re-engineering Finite Element Code

Sequence diagrams are an aspect of UML diagrams that give the sequence of operations that are being programmed [1, 2]. In a simple test of three commercial codes, not named here, it was found that the sequential operations of preprocessing, solving and postprocessing were offered as parallel processes. Thus without preprocessing the problem, the solution could be attempted and without solving the problem postprocessing could be invoked. This is a classic example of engineers focused on developing code that works rather than code that is correct. Such an approach to code writing is




element software

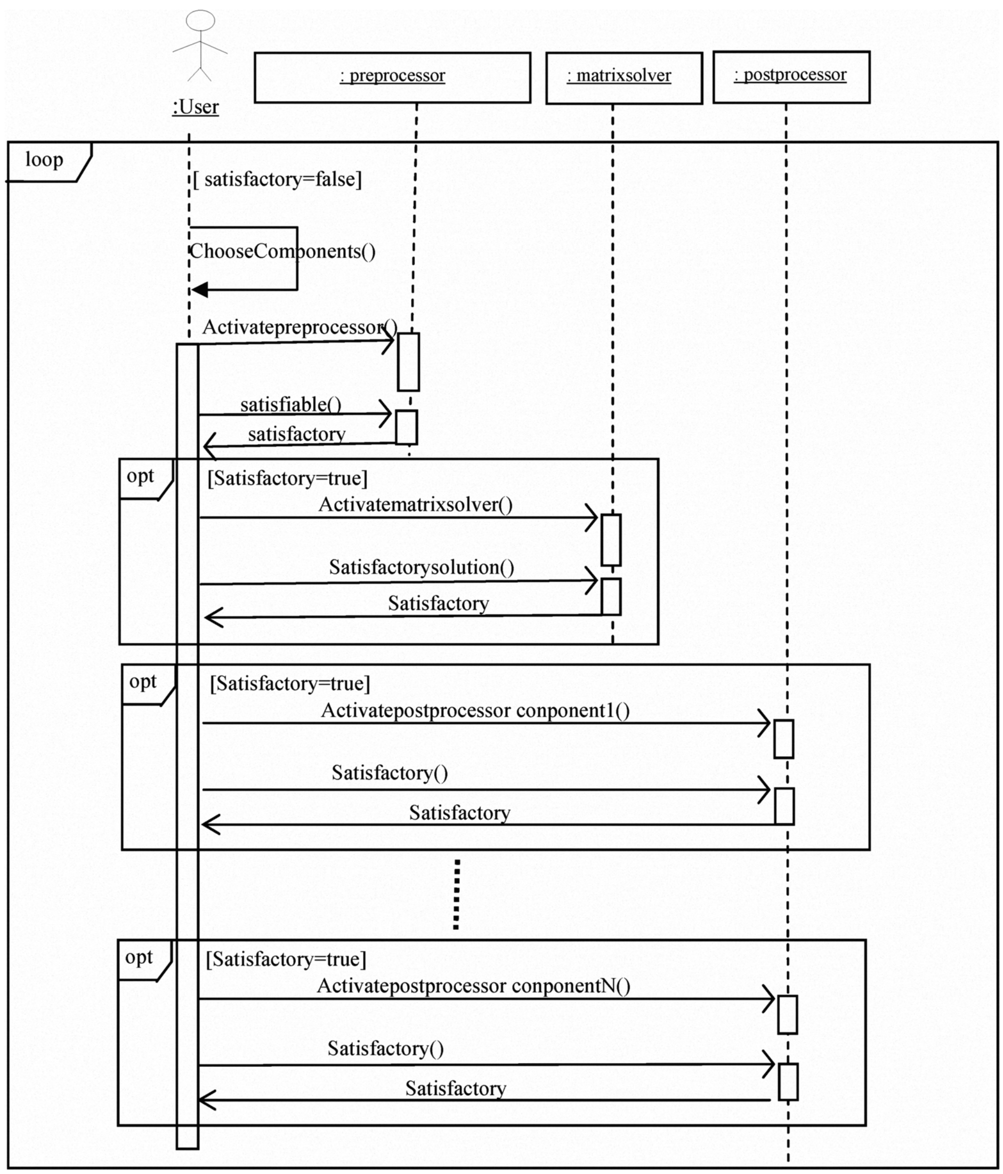



Fig. 10. Sequence Diagram for finite element software

workable but leads to problems when the code needs to be modified or expanded. Further examples include material libraries having to be ready before solution, points being defined before defining lines and lines being defined before defining planes.

Interaction diagrams are used to describe patterns of communication among a set of interacting objects. An interaction diagram is given in one of two forms: sequence diagrams or collaboration diagrams [1, 2]. A collaboration diagram for finite element method software is given in Fig. 9. For the finite element method software, a collaboration diagram is both clear and useful as it is simple, more compact and easily understandable. It is easily noted that it is after preparing finite elements that the matrix solver is activated and it is after solving the matrix that the postprocessor is invoked. At present some commercial software packages are available in which a user can directly activate anything, for example the matrix solver before preparing the finite elements; i.e. without having done the preprocessing properly. In such circumstances, the program gives an error message without crashing but it is an untidy approach.

The sequence diagram shown in Fig. 10 describes the sequence of the operations to be followed in the finite element software. First, suitable components such as finite elements, matrix solver, etc. are chosen. Then the preprocessor is created to do some preprocessing operations during which sometimes it may be impossible to get a proper solution. In that case, the boolean value *false* is returned by the preprocessor. Then components are chosen again and the whole loop is repeated without executing the rest. If the preprocessor returns *true*, then the matrix solver is created. In turn only if it returns *true* that postprocessor1 is created. For example, postprocessor1 might be the equipotential drawing method. The resulting drawing will prove visually to the User whether the solution is good or not. If this stage is passed and postprocessor2, say a force computation method computing force by two different algorithms is used, then it is if the two solutions are the same that the user would be satisfied with the solution; and not if there is a mismatch declared. Likewise it is if all the objects return *true* that the loop is successfully completed. Otherwise, the loop is stopped at the point where *false* is returned and the loop is restarted; i.e. it is if every object returns *true* that the next object is created. It is clearly noted that from Figs. 9 and 10, there will be no activation of methods bypassing a method before that. Therefore, there can be no accidental bypassing activations of method.

## IV. Re-engineering FORTRAN

One of the most important aspects of reverse engineering would involve means of using legacy code written originally in FORTRAN since much of the legacy code is in FORTRAN.

Now facilities are available to convert FORTRAN to C and then C to Java or even FORTRAN directly to Java [16]. The authors tested this by converting programs from FORTRAN to C and C to Java. It is noted that

(a) Classes cannot directly be created from FORTRAN code using forward engineering. Therefore, to make it possible, FORTRAN code should be converted/translated first into Java code and it is thereafter that classes can be created, using reverse engineering.

(b) Earlier, FORTRAN programs were written based on a functional approach to programming rather than an object oriented approach. Therefore even if FORTRAN code is converted to Java code, it will not be in object oriented design.

(c) In the conversion process, a given program is considered as input and it is more or less like an interpreter that translates the given source code into intermediate code; i.e. for example, FORTRAN code is translated to C code where very many statements are generated but the output which displays "Hello WELCOME FORTRAN TO C" is the same. Fig. 11 describes a simple FORTRAN program which is converted into C in Fig. 12. Further, the generated the C program is linked to some library files when it is run. Therefore when the generated C program is converted to Java code, the library files cannot be linked properly from Java platforms. It was found that the problem arises because of input/output operations. Therefore, the C program converted from FORTRAN is modified so that all input/output operations are rewritten to suit the format of C programming language. After this modification, the converted Java program works well. The C program converted from FORTRAN shown in Fig.12 was converted into Java successfully and it is noted that the converted Java program contains 53 lines. Because of the limited space, the converted Java program could not be displayed in this paper. In addition to that it can easily be noted that the FORTRAN program shown in Fig.11 consists only of a single statement excluding the beginning and finishing lines and the converted code in C shown in Fig.12 contains several lines; i.e. a single line in FORTRAN takes many lines in the converted code. Further, only a single class is generated in Java. Therefore auto-code conversion of FORTRAN to Java is not efficient. A finite element program developed in FORTRAN was successfully converted into Java by the authors.

```
PROGRAM TEST
  PRINT*,"Hello WELCOME FORTRAN TO C "
END
```

Fig. 11. FORTRAN Test Program

```
/* Hello.f -- translated by f2c (version 19980831 for lcc-win32).
   You must link the resulting object file with the library:
 libf77.lib
*/
#include "f2c.h"
/* Table of constant values */
static integer c__9 = 9;
static integer c__1 = 1;
#line 3 ""
/* Main program */ MAIN__(void)
{
   /* Builtin functions */
   integer s_wsle(cilist *), do_lio(integer *, integer *, char *, ftnlen),
       e_wsle(void);
   /* Fortran I/O blocks */
   static cilist io___1 = { 0, 6, 0, 0, 0 };
   s_wsle(&io___1);
   do_lio(&c__9, &c__1, "Hello WELCOME FORTRAN TO C ",
(ftnlen)27);
   e_wsle();
   return 0;
} /* MAIN__ */
/* Main program alias */ int test_ () { MAIN__ (); return 0; }
```

Fig. 12. C Code converted from the FORTRAN Test Program of Fig. 11

As Nanjundiah and Sinha [10] have shown with legacy FORTRAN code, in the process of reengineering:

1. Redundant code may be removed, especially in the repetitive application of the same code for computing different quantities.
2. Common blocks, the bane of traditional FORTRAN programming, may be eliminated. Memory allocation for global variables will thereupon be made modular.
3. For optimal memory utilization, the Fortran 90 features of dynamic allocation and deallocation of memory, and pointers (not available with older versions of FORTRAN) may be exploited.
4. Iterative statements using hard-coded numbers (for specifying the range) may be replaced with statements incorporating variables/parameters as the range-specifiers.
5. Conditional statements (IF and GO TO) may be simplified.
6. Names of variables/routines may be made transparent and comprehensible so that their functionality becomes clear. The older version of the code may have used the older FORTRAN standard of utilizing only seven characters to identify a variable which resulted in the variable/procedure names to be terse and cryptic.

In their work, the FORTRAN code was enhanced into another simple form of FORTRAN code and also it was not automated. But in our work reengineering is used to convert source code in C++ into Java and vice versa. In other words, code in one object oriented programming language is converted to another object oriented language. Also our work is automated.

Since FORTRAN is not an object oriented language it cannot be directly reengineered into an object oriented language. Even when we try to activate FORTRAN subroutines via Java, in the reengineering process, it is impossible to generate a class for a FORTRAN subroutine automatically.

## V. Conclusions

This paper has investigated the reengineering of legacy finite element code for purposes of improvement and correction; use of modern object oriented features to obtain the best design and correct code from a software engineering perspective; and the running of the code on the internet by converting the code to Java. As a test, working finite element code authored in C++ has been reengineered to Java. This process recreates the class designs that went into the C++ code, thereby permitting an examination of the design to facilitate improvements to the C++ code itself or alternatively creating Java code using that design. Automatic tools for the conversion are available and have been employed. FORTRAN code needs hardwiring since the concept of class does not exist in FORTRAN. The immediate improvements from old versions of FORTRAN to new versions have been identified from the literature. Changing over to object oriented languages from FORTRAN involves designing the classes.

## Acknowledgement

Mr. Thiruchelvam 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 Rensselaer Polytechnic Institute for facilities to work on his thesis as a Visiting Fellow.

## References

[1] Bernd Bruegge and and H. Dutoit, *Object-oriented Software Engineering – Using UML, Patterns and Java*, 2nd edn., Pearson-Pentice Hall, Upper Saddle River, NJ, 2004.

[2] Sinan Si Alhir, *UML in a Nutshell: A Desktop Quick Reference*, O'Reilly, Cambridge, MA, 1998.

[3] Brad Cox, *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986

[4] S.R.H. Hoole and T. Arudchelvam, "A Formal UML Reliant Software Engineering Approach to Finite Element Software Development for Electromagnetic Field Problems", Proc. *6th Japanese-Mediterranean Workshop on Applied Electromagnetic Engineering (JAPMED06)*, July 2009. Also S.R.H. Hoole and T. Arudchelvam, "A Formal UML Reliant Software Engineering Approach to Finite Element Software Development for Electromagnetic Field Problems," *Journal of Optoelectronics and Advanced Materials* (Under review).

[5] S.R.H. Hoole, *Computer-Aided Analysis and Design of Electromagnetic Devices*, Elsevier, New York, 1989.

[6] Harry G. Schaeffer, *MSC/NASTRAN primer: Static and normal modes analysis*, Schaeffer Systems, 1982.

[7] R.C. Mesquita, R.P. Souza, T. PINHEIRO, and A.L.C.C. Magalhaes, "An object-oriented platform for teaching finite element pre-processor programming and design techniques," *IEEE Transactions on Magnetics,* Vol. 34, p. 3407-3410, 1998.

[8] K.R.C. Wijesinghe, M.R. Udawalpola and S. R.H. Hoole, "Towards object-oriented finite element pre-processors exploiting modern computer technology," *Digests of the IEEE CEFC*, 2002. http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6TGJ-4DTM097-2&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&_docanchor=&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=0fb833de0c7b46d78c769aae708445bc

[9] Ira D. Baxter and Robert L arry Akers, "Component Architecture Reengineering by Program Transformation," *Proc. 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 509-, 2004.

[10] M. Hanna, "Reengineering aims for legacy salvation: migrating code from mainframe computers to new…," *Software Magazine*, Wednesday, September 1 1993.

[11] Ravi S. Nanjundiah and U. N. Sinha, "Impact of modern software engineering practices on the capabilities of an atmospheric general circulation model", *Current Science*, 76 (8). pp. 1114-1116, 1999.

[12] Pedro Sousa and Jürgen Ebert, *Proc. Fifth European Conference on software Maintenance and Reengineering*, IEEE Computer Society Press, 2001.

[13] A. De Lucia, Rita Francese, G. Scanniello and G. Tortora, Developing legacy system migration methods and tools for technology transfer," *Software: Practice and Experience* (John Wiley, NY), Vol. 38, No. 13, pp. 1333-1364, 2008.

[14] K. Kontogiannis, C. Tjortjis and A. Winter (Eds.), Special Issue on the 12th Conference on Software Maintenance and Reengineering (CSMR 2008), *J. of Software Maintenance and Evolution – Research and Practice*, Vol. 21, No. 2, 2009.

[15] http://argouml.tigris.org/. Downloaded May 30, 2009.

[16] http://www.netlib.org/java/f2j/. Downloaded June 5, 2009.

[17] http://www.visual-paradigm.com/product/vpuml/. Downloaded May 30, 2009.