IMPLEMENTATION OF NONLINEAR FINITE ELEMENT USING OBJECT–ORIENTED DESIGN PATTERNS

A. Yaghoobi

Faculty of Mechanical Engineering, University of Tabriz,
Tabriz, Iran

Email: amin.yaghoobi@gmail.com

ABSTRACT

This paper concerns with the aspects of the object–oriented programming used to develop a nonlinear finite element for the analysis of plates based on Reissner–Mindlin theory. To study the shear locking problem in thin plates which occurs in the case of using Full integration method, three kinds of finite elements namely Bilinear, Serendipity and Lagrange with Full, Reduced and Selective Reduced integration methods, are used. By implementing three design patterns of Model–Analysis Separation, Model–UI Separation and Modular Analyzer in the code, the reusability and the extendibility of the program in adding new elements with different number of nodes and integration methods have been increased.

***KEYWORDS**: Nonlinear Finite Element, Object–Oriented Programming, Design Pattern, Reissner–Mindlin Plates*

1.0 INTRODUCTION

Most existing finite element software packages are developed in procedural–based programming languages. These packages are normally monolithic and difficult for a programmer to maintain and extend, though some of them are quite rich in terms of functionality. Extensibility usually requires access to, and manipulation of internal data structures. Due to the lack of data encapsulation and protection, small changes in one piece of code can ripple through the rest of the software system. For example, to add a new element to an existing procedural–based finite element analysis software package, the programmer is usually required to specify, at the element level, the memory pointers to global arrays. Exposing such unnecessary implementation details increases the software complexity and adds a burden to a programmer.

Object Oriented (OO) design principles and programming techniques can be utilized in finite element analysis programs to support better data encapsulation and to facilitate code reuse. A number of object–oriented finite element program designs and implementations have been presented in the literature over the past decade [1–4].

As shown in mentioned papers, the modularity, reusability and extendibility capacities of object–oriented finite element codes are the major characteristics of the approach. The object–oriented methodology has been most successfully applied to various domains of interest in finite element developments. Description becomes easier for algorithms and more natural for basic mathematical equations. Thus, the object–oriented paradigm has been shown to be more appropriate for the easy description of complex phenomena.

Software engineering researchers are developing sets of organizational concepts for designing qualified object–oriented software. These concepts called design patterns. Design patterns in software engineering have been proven to offer great benefits. Especially as engineering software becomes more object–oriented, the importance of design patterns cannot be underestimated.

In this research a set of design patterns for engineering finite element program is implemented using an object–oriented framework in C# to an example. This example is the elastic–plastic analysis of bending plates based on Reissner–Mindlin plate theory. To study the behavior of these plates, three kinds of finite element, namely Bilinear, Serendipity and Lagrange, are used. To overcome the shear locking problem, three integration methods for each element are used to determine the element stiffness matrix. This example shows by applying the design patterns in object–oriented framework, the usability, extensibility, flexibility and maintainability of the code has been increased.

2.0 THE OBJECT–ORIENTED PROGRAMMING APPROACH

A traditional (non–OO) program can be viewed as a logical procedure that takes input data, processes it and returns the output. The main program is built around simpler procedures or functions. In designing a procedural code, one focuses on how to define the logic rather than how to define the data and its organization. In contrast, an OO program is built around objects which encapsulate both the data and the operations on the data. An object can be viewed as an abstraction which relates variables and methods.

Therefore, the first step in building an OO program is to identify the objects and how they relate to each other. Once the objects are identified they can be generalized to a class of objects. A group of objects with the same character is called a class (See **Figure** 1). The software only contains classes. These encapsulate data and data methods. The generic procedures are called methods. The methods represent the behavior of an object and constitute its external interface.

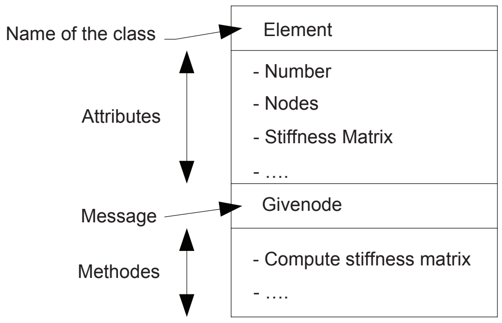


Figure. 1 Class concept.

Object–oriented programming can be said to have four key concepts: abstraction, encapsulation, inheritance and polymorphism. Detailed descriptions of the main concepts of OO programming can be found in many papers [5-7]. Here we merely provide reminders.

Abstraction consists of extracting the most relevant features of the system to be modeled. It provides adequate generalization and eliminates irrelevant details. In OOP, abstraction means to list the defining characteristics of the classes. It also means to state the public interface of the classes, i.e., how their objects will interact with other objects.

The encapsulation concept means hiding the class internal implementation while the class interface is visible. Interaction among objects is controlled by the message mechanism. When an object receives a message, it performs an associated method. The implementation details are not known by the client code. This means that information is hidden outside the class and its derived classes. Information hiding is very useful, for example, if the class public interface is unaltered, the internal implementation can be changed without affecting how the other classes and application programs access that class.

The class information can be specialized using the inheritance principle. Subclasses inherit data and methods of their super–classes. In this way, it is possible to reuse codes in many applications with consequent reduction in development time and costs. The inheritance principle with C# virtual classes introduces an important generalization feature. Pointers to higher level objects of the class hierarchy can represent lower level ones in application programs. This characteristic allows developing type independent code with dynamic binding at runtime. New classes can be added to the hierarchy and the application code will still work with this new type (See **Figure**. 2).

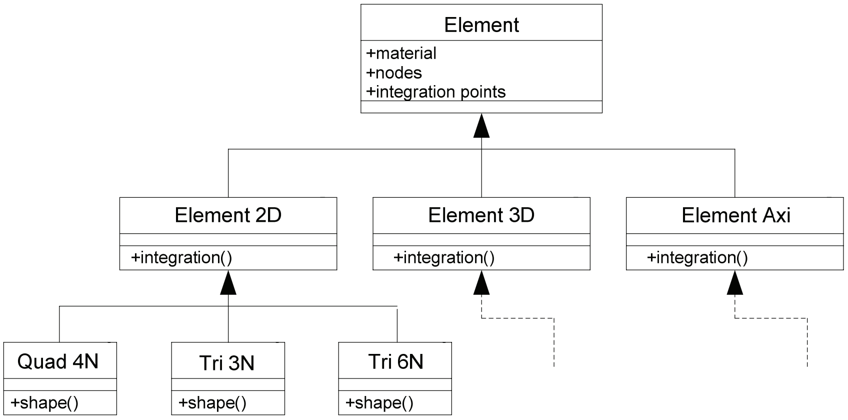


Figure. 2 UML diagram of the element class.

Polymorphism means that objects will answer differently for a same message. For instance, the message "+" may mean concatenation and sum, respectively, for string and matrix classes. Polymorphism and inheritance allow achieving a fairly generic code that selects the methods to be performed at runtime.

3.0 DESIGN PATTERNS

Since object–oriented programming implemented in finite element method for the first time, numerous approaches have been proposed. The design of an OO finite element program is affected by a number of factors, including software requirements, language features, executing environment, etc, that cause to make some differences between the programs. Of course, there are similarities too that reflect consensus among researchers. As this field of research continues to mature, best practices in program design will begin to emerge. It would be useful to capture the key features of these practices in a language–independent and reusable format. Design patterns are a means of achieving this goal. Liu *et.al*. [8] and Fenves *et.al*. [9] explicitly used some of these patterns in their finite element systems. Heng and Mackie [10] used design patterns to identify best practice in object–oriented finite element program design. In this study, three of them will use in a nonlinear finite element method.

3.1. Model–Analysis separation

This pattern decomposes a finite element program into model and analysis subsystems. Rucki and Miller [11] were among the first to explicitly separate analysis classes from the model subsystem. Dubois–Pèlerin and Pegon [12] believed that a clear distinction between analysis–related classes and those related to the model is vital to implementing a flexible program.

3.2. Model–UI separation

The Model–UI separation pattern separates methods and data related to the user interface (UI) from model classes. Most modern finite element systems have integrated graphical user interfaces. In OO finite element programming, a graphical user interface could be implemented by adding UI–related responsibilities to the model classes [13]. In OO finite element programming, the principle of separating graphical classes from model classes was proposed by Ju and Hosain [14].

3.3. Modular analyzer

This pattern decomposes the analysis subsystem into components. Marczak [15] decomposed his analysis subsystem along different lines and also added components representing analysis types, integration schemes, and equation solution algorithms.

In the next sections object oriented finite element implementation will be illustrated using an example of a plate. First, a review of a plate formulation for finite elements in bending– shear based on the theory of Reissner–Mindlin plate is carried out.

4.0 REISSNER–MINDLIN PLATE THEORY

In Reissner–Mindlin plates, normal to the mid–surface (z=0) remains straight but not necessary normal to the mid–surface after deformation [16]. So the effect of shear deformation is considered in this plate unlike Kirchhoff formulation.

4.1. Fundamental relation

The displacement components at a typical point in a Mindlin plate may be represented as:

$$\begin{aligned} u(x,y,z) &= -z\,\theta_x(x,y) \\ v(x,y,z) &= -z\,\theta_y(x,y) \\ w(x,y,z) &= w(x,y) \end{aligned} \tag{1}$$

Where $w(x,y)$ is normal displacement and $\theta_x(x,y)$ and $\theta_y(x,y)$ are the normal rotations of the mid–plane in *xz* and *yz* planes, respectively. The rotations θ_x and θ_y can be expressed in the form:

$$\begin{aligned} \theta_x &= \partial w/\partial x - \gamma_{xz} \\ \theta_y &= \partial w/\partial y - \gamma_{yz} \end{aligned} \tag{2}$$

Where $\partial w/\partial x$ and $\partial w/\partial y$ are the slopes of the middle surface in the *x* and *y* directions and γ_{xz} and γ_{yz} are the shear strains.

4.2. Strain–displacement relations

For Mindlin plate theory, the strain components may be written in terms of the displacements of the middle surface as follows:

$$\varepsilon_b = \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix} = \begin{bmatrix} \partial u/\partial x \\ \partial v/\partial y \\ \partial u/\partial y + \partial v/\partial x \end{bmatrix} = -z \begin{bmatrix} \partial\theta_x/\partial x \\ \partial\theta_y/\partial y \\ \partial\theta_x/\partial y + \partial\theta_y/\partial x \end{bmatrix} = -z \begin{bmatrix} \kappa_x \\ \kappa_y \\ \kappa_{xy} \end{bmatrix} \tag{3}$$

$$\varepsilon_s = \begin{bmatrix} \gamma_{xz} \\ \gamma_{yz} \end{bmatrix} = \begin{bmatrix} \partial u/\partial z + \partial w/\partial x \\ \partial v/\partial z + \partial w/\partial y \end{bmatrix} = \begin{bmatrix} \partial w/\partial x - \theta_x \\ \partial w/\partial y - \theta_y \end{bmatrix} \tag{4}$$

Where ε_b is bending strains vector, ε_s is shear strains vector and κ_x, κ_y , are bending curvatures and κ_{xy} is shear curvature.

4.3. Stress– Strain relations

The moment–curvature relationships and the bending moments are [16]:

$$[\sigma_b] = [D_b][\varepsilon_b] \tag{5}$$

$$[\sigma_b] = \begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} = \int_{-h/2}^{h/2} z \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{xy} \end{bmatrix} dz \tag{6}$$

Where h is the plate thickness and $[D_b]$ is the flexural rigidity matrix for an isotropic material which may be expressed as follows:

$$[D_b] = \frac{Eh^3}{12(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & (1-\nu)/2 \end{bmatrix} \tag{7}$$

Where E is Young's modulus, ν is Poisson's ratio. The shear force–shear strain relationships and also the shear forces are given as:

$$[\sigma_s] = [D_s][\varepsilon_s] \tag{8}$$

$$[\sigma_s] = \begin{bmatrix} Q_x \\ Q_y \end{bmatrix} = \int_{-h/2}^{h/2} \begin{bmatrix} \sigma_{xz} \\ \sigma_{yz} \end{bmatrix} dz \tag{9}$$

Where $[D_s]$ is the matrix of shear rigidity for an isotropic material which may be expressed as:

$$[D_s] = \frac{kEh}{2(1+\nu)} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

Here, k is the shear modification factor and is normally set equal to 5/6 for homogeneous isotropic plates.

4.4. Finite Element Formulation

The normal displacement and normal rotation of the mid–plane at a typical point with local coordinates, (ξ,η) in an element with *n* nodes are obtained using:

$$\begin{bmatrix} w \\ \theta_x \\ \theta_y \end{bmatrix} = \sum_{i=1}^{n} \begin{bmatrix} N_i & 0 & 0 \\ 0 & N_i & 0 \\ 0 & 0 & N_i \end{bmatrix} \begin{bmatrix} w_i \\ \theta_{xi} \\ \theta_{yi} \end{bmatrix} \tag{11}$$

or

$$\vec{u} = \sum_{i=1}^{n} N_i \, \vec{u}_i \tag{12}$$

Where N_i is the shape function corresponded node *i*, and w_i, θ_{xi} and θ_{yi} are the displacement and rotation values of node i. The curvatures in Eq. (3) are expressed as:

$$\begin{bmatrix} \kappa_x \\ \kappa_y \\ \kappa_{xy} \end{bmatrix} = \begin{bmatrix} \partial\theta_x/\partial x \\ \partial\theta_y/\partial y \\ \partial\theta_x/\partial y + \partial\theta_y/\partial x \end{bmatrix} = \sum_{i=1}^{n} \begin{bmatrix} 0 & \partial N_i/\partial x & 0 \\ 0 & 0 & \partial N_i/\partial y \\ 0 & \partial N_i/\partial y & \partial N_i/\partial x_i \end{bmatrix} \begin{bmatrix} w_i \\ \theta_{xi} \\ \theta_{yi} \end{bmatrix} \tag{13}$$

or

$$\varepsilon_b = \sum_{i=1}^{n} B_{bi} \, \vec{u}_i \tag{14}$$

Where B_{bi} is the curvature–displacement matrix. The shear strains in Eq. (4) can be expressed as:

$$\varepsilon_s = \begin{bmatrix} \partial w/\partial x - \theta_x \\ \partial w/\partial y - \theta_y \end{bmatrix} = \sum_{i=1}^{n} \begin{bmatrix} \partial N_i/\partial x & -N_i & 0 \\ \partial N_i/\partial y & 0 & -N_i \end{bmatrix} \begin{bmatrix} w_i \\ \theta_{xi} \\ \theta_{yi} \end{bmatrix} \tag{15}$$

$$\varepsilon_s = \sum_{i=1}^{n} B_{si}\,\vec{u}_i \tag{16}$$

Where B_{si} is the shear strain–displacement matrix. The bending and shear stiffness matrices can be written as:

$$[K_b]^e = \int_A [B_b]^T [D_b][B_b]\,dA \tag{17}$$

$$[K_s]^e = \int_A [B_s]^T [D_s][B_s]\,dA \tag{18}$$

Where $[B_b] = [B_{b1} \;\; B_{b2} \ldots B_{bn}]$, $[B_s] = [B_{s1} \;\; B_{s2} \ldots B_{sn}]$ and $[K_b]^e$ and $[K_s]^e$ are element bending and shear stiffness matrices, respectively. So the total element matrix is:

$$[K]^e = [K_b]^e + [K_s]^e \tag{19}$$

The elemental load vector is also obtained from:

$$F^e = \int_A [N_i]\, q\, dA \tag{20}$$

Where q is the uniformly distributed load. The element Jacobian matrix is:

$$J = \begin{bmatrix} \partial x/\partial \xi & \partial y/\partial \xi \\ \partial x/\partial \eta & \partial y/\partial \eta \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} (\partial N_i/\partial \xi) x_i & \sum_{i=1}^{n} (\partial N_i/\partial \xi) y_i \\ \sum_{i=1}^{n} (\partial N_i/\partial \eta) x_i & \sum_{i=1}^{n} (\partial N_i/\partial \eta) y_i \end{bmatrix} \tag{21}$$

And its inverse is:

$$J^{-1} = \begin{bmatrix} \partial \xi/\partial x & \partial \eta/\partial x \\ \partial \xi/\partial y & \partial \eta/\partial y \end{bmatrix} = \frac{1}{\det J} \begin{bmatrix} \partial y/\partial \eta & -\partial y/\partial \xi \\ -\partial x/\partial \eta & \partial x/\partial \xi \end{bmatrix} \tag{22}$$

Where $\det J$ is the determinant of Jacobian matrix. To obtain shape functions derivative respect to x, y chair rule can be applied, so:

$$\frac{\partial N_i}{\partial x} = \frac{\partial N_i}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N_i}{\partial \eta}\frac{\partial \eta}{\partial x} \tag{23}$$

$$\frac{\partial N_i}{\partial y} = \frac{\partial N_i}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N_i}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{24}$$

The relation between Cartesian coordinate and local coordinate may be written as:

$$dx\,dy = \det J\, d\xi\, d\eta \tag{25}$$

Substituting equation (25) into equation (17) and equation (18), we can obtain:

$$[K_b]^e = \int_{-1}^{+1}\int_{-1}^{+1} [B_b]^T [D_b][B_b] \det J\, d\xi\, d\eta \tag{26}$$

$$[K_s]^e = \int_{-1}^{+1}\int_{-1}^{+1} [B_s]^T [D_s][B_s] \det J\, d\xi\, d\eta \tag{27}$$

4.5. Shear locking

Plate finite elements based on Mindlin theory require only C^0 continuity for displacement and independent rotations, unlike Kirchhoff theory. Therefore, the behavior of the Mindlin plate elements is usually very good for a moderately thick plate situation.

However, when a thin plate is considered, these displacement–based elements cause a problem known as ''shear locking''. When the full integration of the stiffness matrices is used with standard Mindlin finite elements, very stiff results may be obtained in application to thin plates. This means that the bending energy, which should dominate the shear terms, will be incorrectly estimated to be zero in thin plate problems.

To avoid the shear locking problem in thin plates, the reduced and selective reduced integration techniques were proposed in early 1970s [17, 18]. The reduced integration procedure is the reduction in the order of integration in computing the stiffness matrix of the finite element. Similarly, the selective integration procedure is also a kind of reduced integration rule which is used to evaluate the stiffness matrix associated with the shear strain energy. That is to say, this has been adopted to the shear stiffness matrix only and full integration is used on the remaining terms [19]. Therefore, the $[K]^e$ element stiffness matrix can be obtained by separating into bending terms and shear terms. With these definitions, $[K]^e$ element stiffness matrix is given by simplifying the equation (19).

Table 1 shows the full, reduced, and selective reduced integration rules used to test the shear locking response of the Bilinear, Serendipity and Lagrange elements in this paper.

4.6. Plasticity

In the Mindlin plate relations, yield function can be written as a function of bending and shear moments [16]. In this case, we can assume that the whole section of the plate be plastic. Then, we use a yield criterion expressed in terms of bending and shear moments, similar to the Iliushin's yield function [20]. The Iliushin's yield function *F* can be written as:

$$F = \frac{M^2}{M_0^2} - \frac{Y(k)}{\sigma_0^2} = 0 \tag{28}$$

Where *M* are the stress intensities given by:

$$M^2 = M_x^2 + M_y^2 - M_x M_y + 3M_{xy}^2 \tag{29}$$

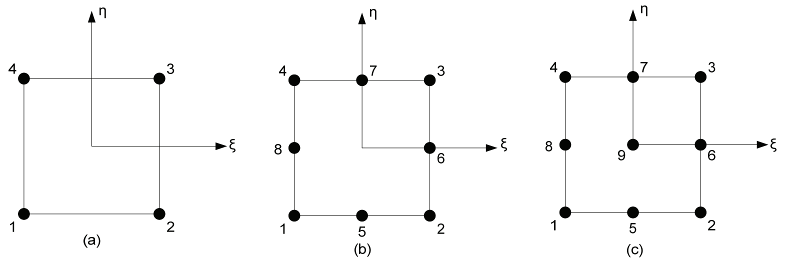
And M_0 is the moment capacity of the cross section. When the cross section is fully plastic, and the moment capacity of the cross section given by:

$$M_0 = \frac{\sigma_0 h^2}{4} \tag{30}$$

The symbol σ_0 is the uniaxial yield stress; $Y(k)$ is a material parameter, which depends on the isotropic hardening parameter *k* and *h* is the thickness of the plate. M_x, M_y and M_xy are stress couples defined by (6).

5.0 APPLICATION

In this study, three design patterns are used to increase the reusability and extensibility of the nonlinear finite element program. For this aim, the elastic–plastic analysis of plates based on Mindlin theory is selected. As shown in section 4.5, to overcome the shear locking problem, the element stiffness matrix is divided in two terms and for each term, different integration method is used. In this paper three kinds of elements which have 4, 8 and 9 nodes, namely, Bilinear, Serendipity and Lagrange, respectively (See **Figure**. 3), and three integration methods are used to analysis the Mindlin's plate (See Table 1). Firstly, *Model–Analysis* separation is used to decompose the finite element program to *Model* and *Analysis* packages. There are essentially two stages in finite element analysis. The first stage involves modeling the problem domain. The second stage involves analyzing the finite element model. It is natural therefore to decompose a finite element program into two major subsystems, one for modeling and the other for analysis. *Model* classes represent finite element entities such as elements, nodes, and degrees–of–freedom (D.O.F.). The *Analysis* subsystem is responsible for forming and solving the system of equations. The two subsystems should be loosely coupled. This means minimizing dependencies across subsystem boundaries. In a procedural code, using many kinds of element with different number of nodes and D.O.Fs persuade the programmer to manipulate the analysis section. But in an object–oriented program, by using the *Model–Analysis* separation pattern, programmer is able to change the element without manipulation of the *Analysis* packages. On the other hand, to implement the different integration method, programmer changes just the *Analysis* packages.



(a)

$$N_i = \frac{1}{4}(1+\xi_i\xi)(1+\eta_i\eta)$$

(b)

At the corner nodes

$$\xi_i = 0, \quad N_i = \frac{1}{2}(1-\xi^2)(1+\eta_i\eta)$$

$$\eta_i = 0, \quad N_i = \frac{1}{2}(1+\xi_i\xi)(1-\eta^2)$$

At the middle nodes

$$N_i = \frac{1}{4}(1+\xi_i\xi)(1+\eta_i\eta)(\xi_i\xi+\eta_i\eta-1)$$

(c)

At the corner nodes

$$\xi_i = 0, \quad N_i = \frac{1}{2}(1-\xi^2)(1+\eta_i\eta)$$

$$\eta_i = 0, \quad N_i = \frac{1}{2}(1+\xi_i\xi)(1-\eta^2)$$

At the middle nodes

$$N_i = \frac{1}{4}(1+\xi_i\xi)(1+\eta_i\eta)(\xi_i\xi+\eta_i\eta-1)$$

At the centre node

$$N_9 = \frac{1}{2}(1-\xi^2)(1-\eta^2)$$

Figure. 3 The finite elements and the shape functions (a) Bilinear (b) quadratic serendipity (c) quadratic lagrange.

Table 1 Integration rules.

	Thick Plate		Thin Plate			
	Full Integration		Reduced Integration		Selective Reduced Integration	
	$[K_b]^e$	$[K_s]^e$	$[K_b]^e$	$[K_s]^e$	$[K_b]^e$	$[K_s]^e$
Bilinear	2×2	2×2	1×1	1×1	2×2	1×1
Serendipity & Lagrange	3×3	3×3	2×2	2×2	3×3	2×2

Figure. 4 shows the packages participating in this pattern and their dependencies. The *Model* package contains model classes, while the *Calculation* and *Solvers* packages together form the *Analysis* subsystem. *Calculation* classes represent different types of analysis. The *Solvers* package consists of mathematical classes for solving system equations. There is no coupling between *Solvers* and *Model*.

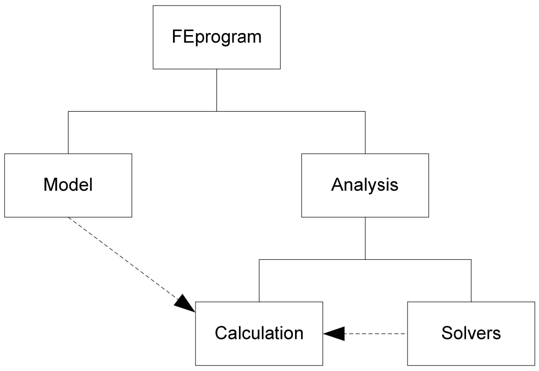


Figure. 4 Packages in the *Model–Analysis separation* pattern.

Since for each element, numbers of nodes are different, FEM classes are also different. But the UI models of elements are the same. Therefore, the *UI–Model separation* pattern is used. UI–related responsibilities should be assigned to UI objects. UI classes should be grouped together in a subsystem that is dependent on the *Model* subsystem. This allows the more volatile UI subsystem to be changed without affecting model classes. Naturally, there should be no coupling between the *Analysis* and *UI* subsystems.

Figure. 5 shows the dependencies between the *UI*, *Mesh* and *FE* packages. *UI* contains classes such as *Struct*, *Curve*, and *Point*. A *Struct* represents a (sub) domain of the finite element model. The *Curve* of a *Struct* describes its boundary. Each *Curve* is in turn defined by its *Point*. These *UI* classes are used to build and manipulate a model on screen. Classes in the *Mesh* package are responsible for generating the mesh based on the on–screen model, creating element and node objects in the process. The class diagram in Figure. 6 shows some *Model* and *UI* classes.

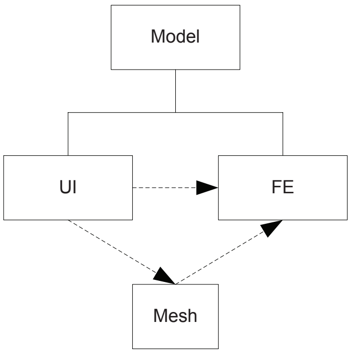


Figure. 5 Packages in the *Model–UI separation* pattern.

In the nonlinear analysis, loading is applied as a linear increment. By using the Modular Analysis pattern, the linear increment is carried out by Elastic package and results involve in the plastic package. This increases the reusability and extensibility of the code, so in other nonlinear analyses, the Elastic packages remain fix.

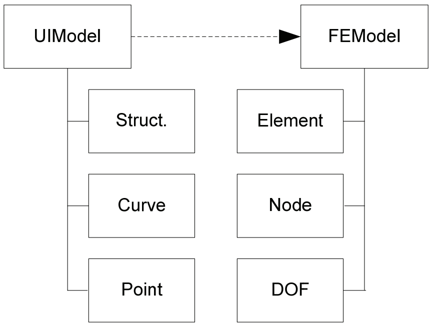


Figure. 6 *Model* and *UI* classes.

Classes that implement Calculation (**Figure**. 7) represent various types of analysis. For example, Static analyzes a finite element model statically. Extending the program to perform, say, transient response analysis can be achieved by implementing a new subtype.

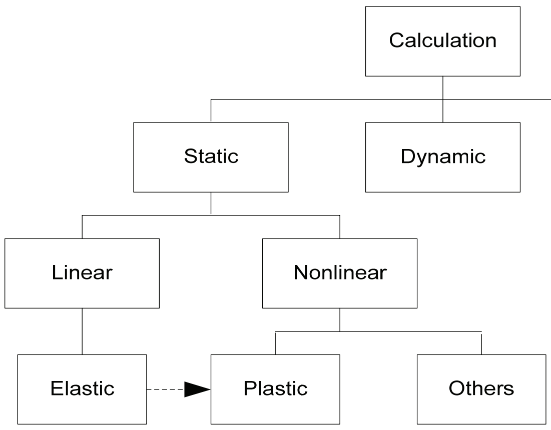


Figure. 7: *Calculation* classes.

6.0 EXAMPLE

The numerical example considered for validation is an isotropic square plate of constant thickness, simply supported on its four sides, subjected to a uniformly distributed load $q = 1.0kPa$. The stress–strain behavior of the plate is elastic–perfect plastic with Young's modulus of $E = 10.92kPa$, Poisson's ratio of $\nu = 0.3$ and yield stress of $\sigma = 1600kPa$. The geometry and material properties are shown in **Figure**. 8.

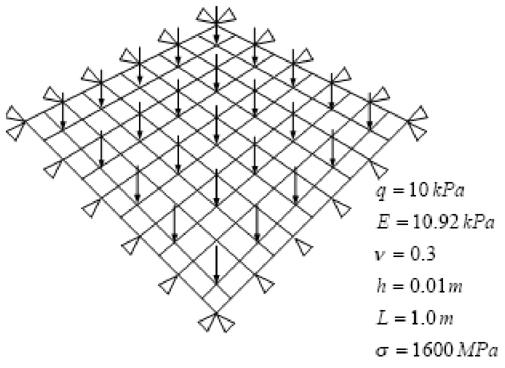


Figure. 8 An isotropic square plate, simply supported on its four sides, subjected to a uniformly distributed load.

We compare our results obtained for Bilinear, Serendipity and Lagrange finite elements using Full, Reduced and Selective Reduced integration methods, with those published by Owen and Hinton [21]. As shown in **Figure**. 9, the Full and reduced Integration methods for Bilinear Element give very stiff results, but Selective Reduced is comparably good.

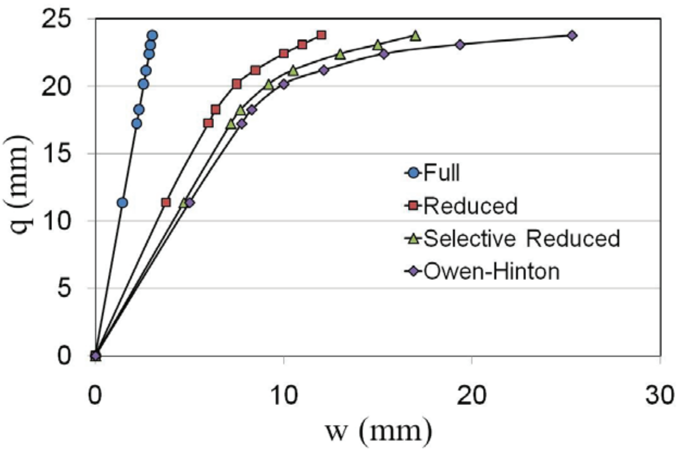


Figure. 9 The load–deflection relation for Bilinear Element.

For analysis of thin plate based on Mindlin theory, as shown in **Figure**. 10 and **Figure**. 11, The Serendipity and Lagrange elements using both Reduced and Selective Reduced Integration methods are suitable.

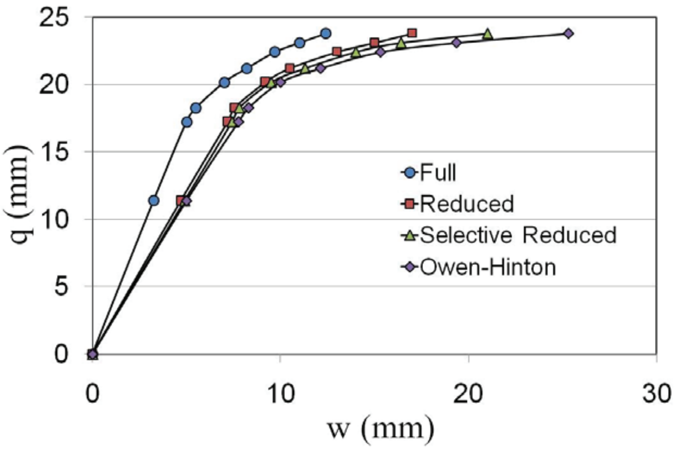


Figure. 10 The load–deflection relation for Serendipity Element.

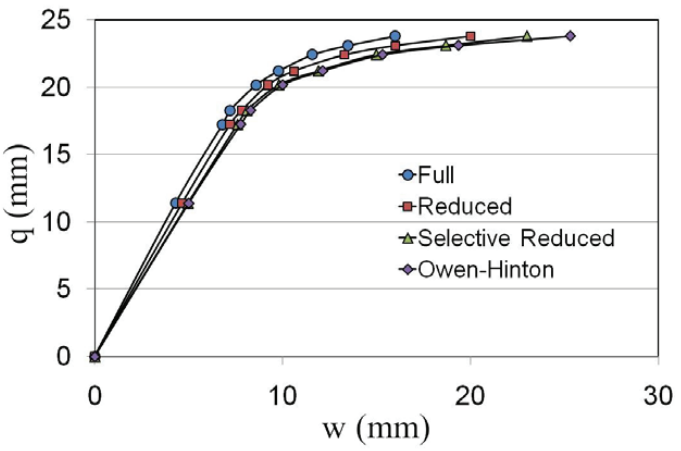


Figure. 11 The load–deflection relation for Lagrange Element.

7.0 CONCLUSION

The Object–Oriented approach is shown to offer undeniable advantages compared to earlier programming structures (procedural based). The encapsulation of the data largely improves the modularity, and thus the reliability and legibility of the code. Inheritance allows an automatic reusability of the already developed methods, and polymorphism is a powerful means to raise the level of abstraction.

Through the example of nonlinear analyzing the Mindlin plates, this paper has shown how using of design patterns into an Object Oriented finite element code, the reusability, extensibility and maintainability of the code increase. In determining of the nonlinear behavior of plates based on Mindlin theory, to access of better accuracy, using element with higher degree of freedoms is proposed. Also, to overcome the shear locking problem, different integration methods are used. Then, the code must be able to add new elements or integration methods. To achieve this aim, three design patterns are used in the Object Oriented code. By using *Model–Analysis separation* pattern, programmer can be able to add new elements to *Model* subsystem and new integration methods to *Analysis* subsystem without any manipulation of the other subsystem. The clear division of responsibilities makes both maintenance and subsequent extensions of the system easier. Also, decomposing the model subsystem to *UI* and *FEM* subsystems help to add new elements to *FEM* subsystems without change the *UI* subsystem. Finally, decomposing the analysis subsystem into components facilitates code reuse without complicating the main hierarchy.

REFERENCES

[1] Miller G. R. An Object–Oriented Approach to Structural Analysis and Design. Computers and Structures 1991;40(1):75–82.

[2] Dubois–Pelerin Y, Bomme P, Zimmermann T. Object–oriented finite element programming concepts. Proceedings of European Conference on New Advances in Computational Structural Mechanics, Elsevier, Amsterdam 1991;95–101.

[3] Dubois–Pelerin Y, Zimmermann T, Bomme P. Object–oriented finite element programming: II. A prototype program in Smalltalk. Comput Methods Appl Mech Eng 1992;98:361–397.

[4] Commend S, Zimmermann T. Object–Oriented Nonlinear Finite Element Programming: A Primer. Adv in Engng Software 2001;32(8):611–628.

[5] Mackie R. I. Object–oriented finite element programming – The importance of data modeling. Advances in Engineering Software 1999;32(9–11): 775–782.

[6] Mackerle J. Object–oriented techniques in FEM and BEM, a bibliography (1996–1999). Finite Elements in Analysis and Design 2000;36:189–196.

[7] Patzak B, Bittnar Z. Design of object oriented finite element code, Advances in Engineering Software 2001;32:759–767.

[8] Liu W, Tong M, Wu X, Lee GC. Object oriented modeling of structural analysis and design with application to damping device conFigureuration. J Comput Civil Eng 2003;17(2):113–22.

[9] Fenves GL, McKenna F, Scott MH, Takahashi Y. An object oriented software environment for collaborative network simulation. In: Proceedings of the 13th world conference on earthquake engineering, Vancouver, Canada; 2004.

[10] Heng B, Mackie R.I. Using design patterns in object–oriented finite element programming. Comput and Struct, doi:10.10161j.compstruc. 2008.04.016.

[11] Rucki MD, Miller GR. An algorithmic framework for flexible finite element based structural modeling. Comput Methods Appl Mech Eng 1996;136:363–84.

[12] Dubois–Pèlerin Y, Pegon P. Improving modularity in object–oriented finite element programming. Commun Numer Methods Eng 1997;13:193–8.

[13] Bettig BP, Han RPS. An object oriented framework for interactive numerical analysis in a graphical user interface environment. Int J Numer Methods Eng 1996;39(17):2945–72.

[14] Ju J, Hosain MU. Finite element graphic objects in C++. J Comput Civil Eng 1996;10(3):258–60.

[15] Marczak RJ. An object–oriented programming framework for boundary integral equation methods. Comput Struct 2004;82:1237–1257.

[16] Mindlin RD. Influence of Rotatory Inertia and Shear on Flexural Motions of Isotropic Elastic Plates. J. Appl. Mech.1951;18(1):31–38.

[17] Zienkiewicz OC, Taylor RL, Too JM. Reduced integration technique in general analysis of plates and shells. International Journal for Numerical Methods in Engineering 1971;3:275–90.

[18] Pawsey SF, Clough RW. Improved numerical integration of thick shell finite elements. International Journal for Numerical Methods in Engineering 1971;3:575–86.

[19] Briassoulis D. On the basics of the shear locking problem of Cv isoparametric plate elements. Comput Structs 1989;33:169–85.

[20] Iliushin A. Plastichnost. Moscow: Gostekhizdat, 1965(in Russian).

[21] Owen D, Hinton E. Finite Elements in Plasticity: Theory and Practice. Swansea: Pineridge Press, 1980.