A new object-oriented finite element analysis program architecture

G.C. Archer\textsuperscript{a, *}, G. Fenves\textsuperscript{b}, C. Thewalt\textsuperscript{b}

\textsuperscript{a}School of Civil Engineering, Purdue University, West Lafayette, IN 47907, USA
\textsuperscript{b}Department of Civil Engineering, University of California at Berkeley, Berkeley, CA, USA

Received 21 January 1998; accepted 22 June 1998

Abstract

This paper presents a new architecture for finite element analysis software, developed using object-oriented design. The resulting system is capable of the modeling and simulation of structural behavior, including the consideration of nonlinear static and dynamic effects. An innovation of the system design is the creation of several classes of objects that separate the analysis tasks from the details of the finite element model. This separation leads to flexible, extensible code. The overall system design and a prototype implementation are presented.

Keywords: Object-oriented; Finite element analysis; Class association map

1. Introduction

Typical finite element programs consist of several hundred thousand lines of procedural code, usually written in FORTRAN. The codes contain many complex data structures, which are accessed throughout the program. This global access decreases the flexibility of the system. It is difficult to modify the existing codes and to extend the codes to adapt them for new uses, models, and solution procedures. The inflexibility is demonstrated in several ways: (1) a high degree of knowledge of the entire program is required to work on even a minor portion of the code; (2) reuse of code is difficult; (3) a small change in the data structures can have a ripple effect throughout the system; (4) the numerous interdependencies between the components of the design are hidden and difficult to determine; (5) the integrity of the data structures is not assured.

Recoding these finite element programs in a new language will not remove this inflexibility. Instead, a redesign is needed. The application of object-oriented design has proven to be very beneficial to the development of flexible programs. The basis of object-oriented design is abstraction. The object-oriented philosophy abstracts out the essential immutable qualities of the components of the finite element method into classes of objects. Objects store both their data, and the operators on the data that may be used by other objects. This abstraction forms a stable class definition in which the relationships between the objects are explicitly defined. The implicit reliance on another component’s data does not occur. Thus, the design can be extended with minimal effort. The abstraction of the data into classes of objects limits the knowledge of the system required to work on the code, to only the class of interest. Encapsulating the data and operations together isolates the classes and promotes the reuse of code. Changes to a class affect only the class in question. There is no ripple effect. Interdependencies between the classes are explicitly laid out in the class interfaces and are easily determined. Object-oriented languages enforce the encapsulation of the classes. The language assures the integrity of the data structures.

* Corresponding author.
In the past several years, many structural engineering researchers have pursued object-oriented design. The current work draws heavily from their efforts. In 1990, Fenves [1] described the advantages of object-oriented programming for the development of engineering software. One of the first detailed applications of the object-oriented paradigm to finite element analysis was published in 1990 by Forde et al. [2]. The authors abstracted out the essential components of the finite element method (elements, nodes, materials, boundary conditions, and loads) into a class structure used by most subsequent authors. Other authors [3–8] have increased the general awareness of the advantages of object-oriented finite element analysis over traditional FORTRAN based approaches.

Several authors present complete object-oriented finite element analysis architectures. Most notably are Miller et al. [9–11], Zimmermann et al. [12–15], Baugh et al. [16, 17], and Jun Lu et al. [18–20]. The coordinate-free geometry presented by Miller relieves the element writers from much of the transformations inherent in element design. Baugh’s creation of coordinate system objects to handle these transformations provides a more traditional approach to transformations. This paper draws from both approaches. Zimmermann’s architecture stresses efficiency of execution. Where possible, the current design stresses efficiency but not at the sacrifice of flexibility. The heart of the system presented by Jun Lu is an assembler object which performs the transformation of element properties between the element coordinates and the structural degrees of freedom (Dof). This concept of a distinct object which separates the analysis from the model is mentioned in brief conference papers by Rihaczek et al. [21], and Chudoba et al. [22]. This separation of tasks is a key concept in the design presented here.

This paper demonstrates a new object-oriented architecture for finite element analysis software. The goal of the system architecture is to provide a flexible and extendible set of objects that facilitate finite element modeling and analysis. The flexibility is achieved by the separation of the finite element analysis tasks into distinct objects. This results in an implementation that is manageable, extendible, easily modified, and provides finite element developers with a system that can be adapted to meet future requirements. The principal innovation of the work is the manner in which the numerical work of the analysis is isolated from the structural model through the use of predefined classes of objects. This isolation limits the degree of knowledge required to work on a component of the system. Although the primary aim of the paper is the presentation of the object design, the implementation of the system is also briefly described with reference to a nonlinear static analysis with Newton–Raphson iteration.

2. Overall architecture

The key concept for the design of the system architecture is the separation of tasks into distinct classes of objects. The primary separation isolates the numerical objects and algorithms of the analysis into an analysis object and isolates the components of the finite element model into a model object. A general overview of the top-level classes of the object design is given in Fig. 1. Class names are shown within boxes, and the associations between the classes are drawn as lines connecting the boxes. The notation is similar to an entity relationship diagram described by Rumbaugh et al. [23], but only a few of the principal classes and associations are shown. At this high level, the system design consists of a model, an analysis, a map, and three handlers.

An analysis object performs an analysis on a model by obtaining the properties of the model object, performing analysis tasks, and updating the model object. The goal of the design is to prevent any dependencies between the model and the analysis objects. This is achieved by the use of the map object, which communicates between the model and the analysis. Model data is expressed in terms of unordered and possibly dependent degrees of freedom. Analysis data consists of ordered and independent unknowns. The map transforms the data, which flow between the two, from one basis to the other.

Also at this high level of the design are three handler objects, so named because they handle the execution of tasks which are model dependent, but must be directed by the analysis. The constraint handler processes the constraints to provide the map with an initial mapping between the model degrees of freedom and the analysis unknowns. The manner in which the constraints are to be handled (e.g. by transformation, Lagrangian multipliers, or penalty functions) is chosen by the analysis, but the handler does all the work. The reorder handler reorders the analysis unknowns according to the criteria chosen by the analysis, typically to reduce the storage requirements of the sparse system property.

![Fig. 1. Top level classes and associations.](image)
matrices. The matrix handler creates and initializes matrices of a specific type by querying the map on behalf of the analysis. The handlers are instantiated by the analysis.

The map object isolates the dependencies between the model components and the analysis data. In fact, much of the difficulty encountered in the modification of finite element analysis programs involves the transformation of model properties from one basis to another. This task has been taken over by the map object. The definition of the map object results in a finite element system that is clear and easy to modify or extend.

3. Functionality of the map class

The map object provides the link between the model and the analysis. In essence, it is a mapping between the degrees of freedom in the model and the unknowns in the analysis. All communication with the analysis is in terms of analysis unknowns, and all communication with the model and its sub-components is in terms of degrees of freedom. The principle function of the map is the transformation of the information that flows back and forth between the model and the analysis.

The map functionality will be introduced using the structural model shown in Fig. 2. The example demonstrates how model properties, in this case an element stiffness matrix, are transformed by the map and given to the analysis, and how the map distributes the results of the analysis to the model. These two tasks are the principal operations of the map. The other methods of the map deal with providing the analysis with access to the collection of objects in the model. These methods are relatively straightforward and are not shown here.

The mapping, or transformation, between the degrees of freedom [shown by means of the coordinate systems (Φr- translational, Φr-rotational) at the nodes] and the three unknowns in the analysis (symbolically represented as dotted lines) is determined by the constraint handler and stored by the map object. The mapping for the sample problem is shown in Table 1. The mapping consists of DOF description objects, indicating the degree of freedom component in the model, and pairs of analysis unknowns and transformation values. For the sample problem, the maximum number of terms is 2. Obviously this length will vary from model to model. Nodes A and D have been constrained out (i.e. no analysis unknown is assigned to them) and node B is slaved to node C. These transformation values give the complete transformation between the model degrees of freedom and the analysis unknowns.

3.1. Transformation of a stiffness matrix

To demonstrate the use of this mapping, consider the case of the transformation of the stiffness matrix for element A. Element A will internally produce its stiffness matrix $k_e$ according to the elemental system. It also generates a transformation matrix $T^e_{r-f}$ that will transform the matrix into the terms of the local coordinate systems at the nodes. The map defines a transformation matrix $T^m_{m-n}$ to put the matrix in terms of the actual nodal coordinate systems that describe the degrees of freedom, and an additional transformation matrix $T^n_{n-a}$ to put the matrix in terms of the analysis unknowns. Algebraically, this process of producing $K_{na}$, an element stiffness matrix in terms of analysis unknowns, is represented as:

$$K_{na} = T^m_{m-n} T^e_{r-f} k_e T^n_{n-a}.$$

For this example, $k_e$ is a $3 \times 3$ symmetric matrix, and $T^e_{r-f}$ is a block matrix with a single row block and four column blocks, $r$, corresponding to the local co-
ordinate systems as shown in Fig. 3(a). The two subscripts for the column blocks, for example $t_x$, represent the coordinate system transformed from (in this case the elemental system), and transformed to (in this case the translational coordinate system for node I), respectively.

The map object queries the augmented stiffness matrix, $T_{e...}$, produced by the element, to determine these local coordinate systems. To transform the matrix into the nodal coordinate systems that define the degrees of freedom, the map produces $T_{l...n}$, which is a 4x4 block matrix. Only the diagonal blocks contain transformations. These transformations are obtained from the comparison between the local coordinate systems at the nodes defined by the elements, and the coordinate systems defined at the nodes. The coordinate system objects themselves actually produce the transformation blocks. The number of axes of the coordinate systems determines the size of the blocks. In this example, the coordinate systems from the element and at the node have the same number of axes, resulting in square blocks. The coordinate systems at the nodes and the transformation matrix the map adds for the local-to-nodal transformation are shown in Fig. 3(b).

Finally, the map must add the transformation $T_{a...a}$ to put the stiffness matrix into the terms of the analysis unknowns. The values for this transformation are obtained from the degree-of-freedom-to-analysis-unknown mapping shown previously in Table 1. The map first creates a list of all the analysis unknowns affected by the augmented stiffness matrix. These analysis unknowns are mapped to the reordered analysis unknowns, and become the labels for the columns of $T_{a...a}$. The rows of $T_{a...a}$ are blocked according to the columns of $T_{l...n}$. In this case, $T_{a...a}$ has four row blocks and one column block as shown in Fig. 3(c). Each column in $T_{a...a}$ refers to a specific analysis unknown (in this case columns 1, 2, and 3 refer to unknowns 1, 2, and 3, respectively).

When all of the transformations are applied, the resulting stiffness matrix will be a 3x3 matrix that refers to analysis unknowns 1, 2, and 3. A similar procedure is employed by the map to transform the mass, and damping matrices, and the resisting force and applied load vectors.

### 3.2. Processing of responses

The map receives the response arrays (e.g. displacements, velocities, accelerations, etc.) from the analysis, and must distribute them to the nodes. The arrays are, in terms of analysis, unknowns. The map must transform these response arrays to a model degree of freedom basis. For each degree of freedom in the model, the map obtains the associated analysis unknown results and transforms them using its internal degree-of-freedom-to-unknown-number mapping scheme. Referring to Table 1 of the previous example, the map may be given a displacement array, $d$, by the analysis, for distribution to the nodes. The map then processes each degree of freedom at each node and calculates the effect of $d$ on the degree of freedom. For node A, there is no effect, as both the translational and rotational degrees of freedom have no associated analysis unknown. The next degree of freedom the map encounters is the translation degree of freedom for node B. The horizontal axis component corresponds to the response value of unknown number 1 ($d_1$). The vertical axis component is then calculated as $d_2 - 5.0 d_3$. The axis components are grouped into vectors using the coordinate system at the nodes. The

### Table 1

Mapping of degrees of freedom to analysis unknowns for the model in Fig. 2

<table>
<thead>
<tr>
<th>Node</th>
<th>DOF Type</th>
<th>Axis</th>
<th>Term 1</th>
<th>Term 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>translation</td>
<td>x</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>translation</td>
<td>y</td>
<td>1</td>
<td>-5.0</td>
</tr>
<tr>
<td>B</td>
<td>rotation</td>
<td>xx</td>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>C</td>
<td>translation</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>translation</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>translation</td>
<td>x</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>rotation</td>
<td>xx</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>DOF Type</th>
<th>Axis</th>
<th>Unknown $\xi$</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>translation</td>
<td>x</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>B</td>
<td>translation</td>
<td>y</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>C</td>
<td>rotation</td>
<td>xx</td>
<td>1</td>
<td>0.3</td>
</tr>
</tbody>
</table>
resulting vector describes the effect of \( d \) on the translation degree of freedom of node B. It is then used to update the node. The remaining degrees of freedom for the model are processed similarly.

4. Functionality of the handler classes

The object design contains three classes of handlers, which are used by the analysis to manage the more complex tasks that are shared between the many different classes of analysis objects. These are the constraint handler, the reorder handler, and the matrix handler. The constraint handler manages the processing of the constraints, the reorder handler manages the ordering of the analysis unknowns according to analyst specified guidelines, and the matrix handler is responsible for providing a specified type of matrix and vector, initialized according to the sparsity of the equations. The handlers not only isolate their tasks from the rest of the system, they encourage the reuse of significant amounts of code. Writers of analysis objects simply choose from the existing set of handlers or add a new handler to the library for all to use.

4.1. Constraint handler

The constraint handler is responsible for processing the constraint objects and providing the map object with an initial mapping between the model degrees of freedom and the analysis unknowns. Conventional constraint handlers such as transformation, penalty function, and Lagrangian multiplier are provided. New handlers can be created as needed by a particular method of analysis.
Collectively, constraints in the finite element model appear as homogeneous multi-point constraint equations in the form of:

\[ C \hat{q} = 0. \] (2)

Where \( \hat{q} \) represents a vector of the degree of freedom components in the model, \( \theta \) is a vector of zeros, and \( C \) is a sparse matrix of constants that represents the relationship between the degree of freedom components.

In the transformation constraint handler, the constraint matrix \( C \) is converted by the handler to a transformation matrix that relates all the degrees of freedom to a constrained set of degrees of freedom. The constrained set of degrees of freedom are numbered, and this initial mapping between the model degrees of freedom and the analysis unknowns is passed to the map object. These transformations are applied when transforming element stiffness into those needed in the final set of analysis unknowns.

For the penalty function constraint handler, the initial mapping between the model degrees of freedom and the analysis unknowns is one-to-one, with no transformation. The constraints are satisfied by the addition of the term, \( C^T \alpha C \), to the stiffness matrix. The preparation of the penalty matrix, \( \alpha \), the multiplication of the additional term, and its inclusion in the stiffness matrix is the responsibility of the analysis object.

For the Lagrangian multiplier constraint handler, the initial mapping between the model degrees of freedom and the analysis unknowns is one-to-one, with no transformation, but the handler sets up the additional set of analysis unknowns. These additional analysis unknowns represent the Lagrangian multipliers. The analysis object is responsible for augmenting its stiffness matrix with the constraint matrix to satisfy the constraints.

4.2. Reorder handler

The reorder handler is responsible for providing the map object with a reordered mapping between the model degrees of freedom and the analysis unknowns based on criteria provided by the analysis object. To accomplish this task, the reorder handler requires complete model information.

The internal workings of the reorder handler object are unique to the type of reordering scheme the object represents. In general, the reorder handler will obtain lists of elements, nodes, and displacement boundary conditions from the model object. Also, it will obtain from the map object the initial mapping of the analysis unknowns. The reorder handler is then responsible for providing the map object with a reordered set of analysis unknowns.

4.3. Matrix handler

The matrix handler is responsible for providing matrices and vectors of a specific type, initialized according to the connectivity of the model. Each type of matrix and vector will have its own matrix handler class. The choice of matrix handler by the analysis object defines the type of matrix or vector it will receive.

The internal workings of the matrix handler object are unique to the type of matrix and vector the object creates. In general, the matrix handler will query the map object to determine the connectivity of the analysis unknowns. For example, the matrix handler for the column compacted matrix type will need to know the smallest analysis unknown connected to each analysis unknown to define the column heights for the matrix.

5. Functionality of the model classes

The finite element model is represented in the object design as a model object. The model object contains collections of its constituent objects and provides access to these collections as requested. The entity relationship diagram for the model object is shown in Fig. 4. The diamond symbol indicates that the model object is merely an aggregation of its component objects. The darkened circle indicates the multiplicity of the association. Thus, a model object may have many nodes, elements, load cases, boundary conditions, and constraint objects. Collectively, the purpose of the model objects is to track the state of the finite element model as the analysis progresses, and report on the state when requested.

The abstraction of the finite element tasks into objects flows from the natural division of duties of the finite element method. The basic responsibilities of elements, nodes, and load case objects are obvious. The finer points are discussed later in this section.

Initially, it appears redundant to have both constraints and boundary conditions. A constraint object represents a homogeneous multi-point constraint equation between degrees of freedom. A boundary
condition object, on the other hand, is single-point and may vary in time. The prescribed values for the boundary condition objects are stored in the load case objects. This distinction between boundary conditions and constraints is made to allow for efficient treatment of time-varying constraints in the analysis.

5.1. Nodes

The node object represents the traditional view of a node in the finite element method. That is, a node has a position in space, defines a set of degrees of freedom, and tracks the state of its degrees of freedom. The entity relationship diagram for the node object is given in Fig. 5. The triangular symbol indicates that the dof class is the superclass of the vectordof and scalardof classes. That is, a vectordof or scalardof object may be used wherever a dof object is required.

The map relies on the nodes to provide coordinate system objects that represent the orientation of the degrees of freedom at the node. The degrees of freedom are vector based and must be updated by providing the node with response vectors. It is the map's responsibility to assemble these vectors. Elements query the state of the degrees of freedom in order to advance their own state. Once again, this communication is vector based, the element is free to query the response vectors in any convenient coordinate system.

5.2. Elements

The primary responsibilities of an element are to provide the map with the current linearized stiffness, mass, and damping, and the current resisting force. The entity relationship diagram for the element object is given in Fig. 6. An actual element object is a subclass of the abstract element class. Each member object represents a specific type of finite element. The elements connect to nodes, and from them obtain the current state of the model. As noted in the map example, elements report their current state to the map in terms of a property matrix augmented with the transformations that place the matrix in the basis of coordinate system objects. The element is free to choose the basis in a manner similar to that given in Refs. [10, 16]. Thus, elements are free from using specific assumed coordinate systems at the nodes. For example, a truss bar can be defined in 1D, but exist correctly in a 2D or 3D model.
Elements make use of constitutive models of specific materials, from which the required action–deformation relationships (e.g., stress–strain, moment–curvature, etc.) can be obtained. Element load objects that are specific to each type of element can load elements. The element load objects define the magnitude and type of load. The effects of the load are computed by the element itself, and are included in the resisting force calculation. An element can also be given an initial state using an initial element state object. These effects are also included in the resisting force calculation. The element load and initial element state objects are stored in load cases.

5.3. Load cases

A load case consists of a collection of loads, prescribed displacements, and initial element state objects. These objects communicate directly with the map to provide their current state. The entity relationship diagram for the load case object is given in Fig. 7. The load objects may be either nodal loads or element loads. In either case, the load vector is given to the map object augmented with the transformations and coordinate systems used to represent the load vector. Nodal loads may be related to either scalar or vector degrees of freedom at a node. Element loads are related to the element to which they apply, and are only responsible for providing the element with the magnitude and type of element load. The element itself performs all calculations. A prescribed displacement object provides the time varying value for a boundary condition in the model. Initial element states contain the parameters that describe the initial state of a specific instance of a type of element. A load case object maintains the collections of loads, prescribed displacements, and initial element state objects under its control, and provides access to these collections to the analysis object, via the map.

6. Utility objects

Several classes of utility objects are necessary to facilitate both the modeling and analysis tasks. First, a set of numerical objects including matrices and vectors are required. A set of geometrical objects is necessary to provide support for geometric points, vectors, and coordinate systems. Finally, to facilitate the transformation of model properties, a class of objects consisting of a property matrix or vector, augmented with pending transformations is needed.

6.1. Numerical objects

The design of an object-oriented library of numerical objects is best left to specialists in numerical analysis [15, 24]. This frees engineers to concentrate on the structural aspects of the software design. The aim of the current research is to identify the numerical needs of the finite element analysis design. The selection of a suitable numerical library is deferred for future work. Due to the wide use of the FORTRAN coded LAPACK library [15], a good candidate is the C++ based LAPACK++ library [24].

The current research has identified two types of users of the numerical objects, small-scale and large-scale. The map and model objects are small-scale users. They use the numerical objects to form and transmit element and nodal properties. These objects
are small, generally full, possibly symmetric, and possibly contain blocks of zero or identity entries. The principal operations performed on these objects are addition, subtraction, transposition, and multiplication. The analysis object, on the other hand, is a large-scale user. It may create large matrices with highly specialized sparsities. These may include symmetric banded, column compacted, upper or lower triangular, general sparse, etc. In addition to the basic algebraic operations, the analysis object requires these matrix objects to efficiently perform complex numerical operations such as the solution of linear equations and the eigenvalue problem. To initialize these objects, the analysis uses the matrix handler associated with the type of matrix or vector needed. The handler provides an initialized matrix to the analysis.

6.2. Geometrical objects

The geometrical objects are responsible for providing the means to represent and manipulate points, vectors, and coordinate systems. The relationship between the objects is shown in Fig. 8. As previously demonstrated, the model properties and state information are vector based. The basis of the vectors is transformed via rotation matrices. In a manner similar to that given in Refs. [10, 16], the coordinate system objects, in comparison with other coordinate systems, provide the rotation matrices. The benefit of this vector-based approach is that it isolates the generation of transformation matrices into one class of objects. Namely subclasses of the coordinate system object. This promotes efficient coding of these primary objects. By making all subclasses of the coordinate system class able to transform to and from a 3D rectangular coordinate system, the set of defined coordinate system objects can be easily extended. This permits existing code for the elements, the map, and the handlers to use new classes of coordinate system objects without modification.

6.3. Augmented matrix objects

As demonstrated in the map example, model properties and state information are transmitted to and from the analysis via matrices augmented with pending transformations [as shown in Eq. (1)]. Thus, a class of augmented matrix objects is provided. Their basic task is to store the base property matrix, store and report on the pending transformations, and efficiently produce the resulting transformed property when requested.

The main benefits to isolating this task in one object are the promotion of efficient coding and the enforcement of the reuse of code. To reduce the number of computations, it is advantageous to combine the pending transformations into one, before performing the matrix triple product. Furthermore, the transformations are stored in a blocked structure to take advantage of zero and identity blocks that are common in transformations. The block structure flows naturally from the vector based nodal and element properties.

7. Implementation

The validity of the object model was confirmed by implementation in the C++ programming language. The details of the implementation are given in Ref. [25]. To provide a brief introduction to both the program and the analysis object, the implementation of a sample nonlinear static analysis with Newton–Raphson iteration (the FeaNRStaticanalysis object) is now provided. This analysis scheme involves removing the unbalance in the equilibrium equations by continually advancing the state of the model, using the current tangent stiffness and the unbalanced load, until equilibrium is reached.

The FeaNRStaticanalysis class consists of an interface, which defines both the public methods available to users of the class, and the private details of the implementation; a constructor, which provides an instance of the class; an init method, which initializes the analysis; and an analyze method, which performs the analysis. The implementation of the class is shown in a simplified C++ code, with the memory management, pointers, references, and some data typing information removed for clarity. The interface for the FeaNRStaticanalysis class is shown in Fig. 9.

The constructor for the FeaNRStaticanalysis class creates: a constraint handler, which processes the constraints of the model using the method of transformations; a reorder handler, which minimizes the bandwidth of the stiffness matrix; a map, which is used to communicate the properties of the model; and a matrix handler, which provides initialized column compacted matrices and full vectors. The constructor also
initializes the stiffness matrix and the vectors that hold the applied load, resisting force, unbalanced force, and displacement. The implementation for the constructor is given in Fig. 10.

Three private assembly methods (`assembleLoad`, `assembleResistingForce`, and `assembleStiffness`) are constructed to aid in the analysis. Each iterates through components of the model and in turn requests a property array from the map, and assembles them into the system array held by the analysis. For example, implementation for the `assembleStiffness` method is given in Fig. 11.

An iterator for the elements, `elementItr`, is obtained from the map. The iterator represents each element in turn, and is given as the argument to the `getStiffness` method of the map. The resulting stiffness matrix, $K_{el}$, is then assembled into the system stiffness matrix, $k$, using the `assemble` method of the system stiffness matrix. The `assemble` method takes two arguments, the transformed element stiffness matrix obtained from the

```java
class FeaNRStaticAnalysis{
    // private data
    string loadCase; // load case
    FeaMap map; // map
    FeaMatrix k; // stiffness matrix
    FeaVector unbal; // unbalanced load vector
    FeaVector rf; // resisting load vector
    FeaVector d; // displacement vector
    FeaVector p; // load vector
    // private methods
    void assembleK(); // assembles stiffness
    void assembleTotalLoad(double lsf); // assembles total load
    void assembleResForce(); // assembles resisting force

    public:
    FeaNRStaticAnalysis(FeaModel model) // constructor
    void init(string loadCase); // initialize analysis
    void analyze(double loadStepFactor) // analyze LC up to factor
}
```

![Fig. 9. Interface for the Newton–Raphson static analysis class.](image)

![Fig. 10. Constructor for the Newton–Raphson static analysis class.](image)
The **getTransformedMatrix** method which applies pending transformations, and an identification array of associated analysis equation numbers obtained from the **getID** method of the augmented array.

The initialization method is responsible for setting the load case and initializing the state of the elements. The **analyze** method advances the analysis up to the given **loadStepFactor**. The implementation for the **analyze** method (with non-converging loop control not shown) and the implementation for the **init** method are given in Fig. 12.

In the first three lines of the **analyze** method, the applied load, resisting force, and unbalanced load are obtained. The norm of the unbalance load is then tested against a small tolerance to check if equilibrium has been obtained. If not, the displacements due to the unbalanced load are calculated. This displacement increment is added to the nodes by the map object by invoking the **updateNodes** method of the map with the displacement increment as the method's argument. The state of the elements is then updated by using the map's **updateElements** method. This update is tentative until it is committed. The resisting force and unbalanced load are calculated, and execution returns to the top of the whole loop. If the norm of the unbalance is sufficiently small, execution stops, otherwise the loop is repeated. The equilibrium state is then committed by the map object and becomes part of the solution path.

To analyze a structure, the model and loading are first defined. The analysis is then directed using the code shown in Fig. 13. As can be seen, the code for analysis can be described at a very high level due to the services provided by the map and the handlers. In

```
void init(string lc) {
  // initializes the analysis
  loadCase = lc;
  map.initializeElSt(loadCase); // initialize the element states
}

method analyze(loadStepFactor) {
  // analyze for the given load case
  p = assembleLoad(loadStepFactor) // form applied load
  rf = assembleResistingForce() // form resisting force
  unbal = p - rf // calculate force unbalance
  while (unbal.norm2() > TOL){ // loop to remove unbalance
    k = assembleStiffness() // form stiffness matrix
    d = d + solve(k, unbal) // update displacements (d)
    map.updateNodes(d) // update nodes
    map.updateElements() // update elements
    rf = assembleResistingForce() // update resisting force
    unbal = p - rf // calculate force unbalance
  }
  map.commit() // commit current state
}
```

Fig. 12. Initialize and analyze methods for the Newton–Raphson static analysis class.
fact, the analysis scheme essentially just directs the map to perform the appropriate tasks.

The increased flexibility of the object-oriented software architecture impacts the execution speed of the program. To investigate the extent of the slowdown, the program was compared to an existing FORTRAN based research and education program (DRAIN-2DX ver. 1.03 Ref. [26]). The sample problem consisted of 100 steps of a push-over analysis of a 7-bay, 7-story, steel frame structure modeled using inelastic beam-columns. Great care was taken to ensure that both programs were performing similar calculations and storing similar amounts of data. On a Pentium-Pro 200, the object-oriented program took 56 s while DRAIN-2DX took 45 s. The FORTRAN code was about 20% faster. The slowdown is minor and perhaps due in part to the young age of the object-oriented program (it is not yet bound to a fast numerical library). Also contributing to the program slowdown is the large number of function calls and dynamic bindings inherent in object-oriented programming.

8. Conclusions

An object-oriented design for a finite element program has been presented. The design was implemented in the C++ programming language. The program is flexible, extendible, and easily modified for a variety of finite element analysis procedures. The problems inherent in procedural based finite element programs are eliminated by design. To modify or extend the program, the required knowledge of the components is restricted to the public interfaces of the classes. The reuse of code is promoted by the use of inheritance. The effect of a change to the data structures is limited to the class that owns the data; there is no ripple effect through the program. Dependencies between components of the design are explicitly defined and are easily determined from the public interfaces of the classes. The encapsulation features of the implementation language enforce the integrity of the program’s data structures.

The principal feature of the software architecture is the map object. The map isolates the degree-of-freedom based information in the model from the governing equation information in the analysis. It is this isolation that limits the degree of knowledge required to work on a component of the system, and provides the flexibility in applications. As a result of the creation of the map object, writers of analysis objects do not need details of the model and can focus on the numerical algorithms. Also, elements can be formulated with reference to any convenient coordinate system, the map handles the transformations. Finally, the orientation of the degrees of freedom can vary. The map makes no assumptions as to the coordinate systems used at the nodes.

The architecture includes three handlers, each designed to perform tasks that are common to most analysis methods. This greatly reduces the workload of authors of analysis objects. The handlers are: the constraint handler, which processes the constraints; the reorder handler, which reorders the equations of equilibrium for efficiency; and the matrix handler, which constructs the matrices and vectors used by the analysis to hold the system properties. The handlers not only isolate their tasks from the rest of the system, they encourage the reuse of significant amounts of code. Writers of analysis objects simply choose from the existing set of handlers or add a new handler to the library for all to use.

References


```c++
FeaNRStaticAnalysis analysis(model);
analysis.init(“loadCase1”);
for lsf = 0.0 to 1.0 by 0.05 {
    analysis.analyze(lsf);
    outputNodesAndElement();
}
```

Fig. 13. Use of the Newton–Raphson static analysis class.


