Nonlinear Finite-Element Analysis Software Architecture Using Object Composition

Frank McKenna[1]; Michael H. Scott[2]; and Gregory L. Fenves[3]

Abstract: Object composition offers significant advantages over class inheritance to develop a flexible software architecture for finite-element analysis. Using this approach, separate classes encapsulate fundamental finite-element algorithms and interoperate to form and solve the governing nonlinear equations. Communication between objects in the analysis composition is established using software design patterns. Root-finding algorithms, time integration methods, constraint handlers, linear equation solvers, and degree of freedom numberers are implemented as interchangeable components using the *Strategy* pattern. The *Bridge* and *Factory Method* patterns allow objects of the finite-element model to vary independently from objects that implement the numerical solution procedures. The *Adapter* and *Iterator* patterns permit equations to be assembled entirely through abstract interfaces that do not expose either the storage of objects in the analysis model or the computational details of the time integration method. Sequence diagrams document the interoperability of the analysis classes for solving nonlinear finite-element equations, demonstrating that object composition with design patterns provides a general approach to developing and refactoring nonlinear finite-element software.

DOI: 10.1061/(ASCE)CP.1943-5487.0000002

CE Database subject headings: Computer programming; Computer software; Finite element method; Nonlinear analysis.

Author keywords: Computer programming; Computer software; Finite element method; Nonlinear analysis.

Introduction

Performance-based methodologies in structural engineering have increased the need for high-fidelity simulation of structural response under extreme loads, such as earthquake, blast, and other events that may cause damage or lead to progressive collapse (Moehle and Deierlein 2004). Simulation software for performance-based engineering must be able to accommodate sophisticated constitutive models for conventional and novel materials and soils, large displacement analysis methods, and robust solution algorithms for dynamic loads, among many other requirements. The finite-element method provides a general methodology for simulating the response of structural and geotechnical systems to arbitrary loading. To incorporate future developments and specific user needs, simulation software must provide interfaces for new finite-element formulations, solution algorithms, equation solvers, and support for advanced computing, modeling, visualization, and data mining. For example, parallel computing is becoming common in engineering, and structural simulation software needs to be able to take advantage of hardware systems that range from multicore processors to massively parallel computers (Modak and Sotelino 2002; Peng et al. 2004).

To address these requirements, finite-element simulation software must be designed for computational efficiency, flexibility, extensibility, and portability. The traditional focus of simulation software development has been efficiency, but the other goals are equally important when considering the complete software lifecycle. Flexibility means that software components can be combined to provide new capability, even if it was not anticipated in the original design. Extensibility means that both the design and implementation of software components can be made more specific or to provide additional functionality. Portable software is designed to run on a variety of computer architectures and operating systems to take advantage of new computing capability.

To address these needs, this paper presents a new object-oriented architecture in which the goals of flexibility, extensibility, and portability of finite-element software are achieved by emphasizing object composition over implementation inheritance in the software design. The major contribution is the use of composition of software components that implement solution procedures for the nonlinear governing equations of a finite-element model. Object composition is shown to provide a superior software design compared with the more common use of class inheritance. In addition to composition, the software architecture uses software design patterns to organize communication between the components of a nonlinear finite-element analysis. The architecture allows these components to be combined to create customized simulation applications, further enhancing flexibility, extensibility, and portability.

The modular nature of the finite-element method results from its mathematical formulation (Hughes 1987; Bathe 1996; Zienkiewicz and Taylor 2005). Several researchers have developed object-oriented software designs and implementations for

[1]Research Engineer, Dept. of Civil and Environmental Engineering, Univ. of California, Berkeley, CA 94720. E-mail: fmckenna@ce.berkeley.edu

[2]Assistant Professor, School of Civil and Construction Engineering, Oregon State Univ., Corvallis, OR 97331 (corresponding author). E-mail: michael.scott@oregonstate.edu

[3]Dean, Cockrell School of Engineering, Univ. of Texas at Austin, Austin, TX 78712. E-mail: dean@engr.utexas.edu

Note. This manuscript was submitted on April 21, 2008; approved on November 2, 2008; published online on December 15, 2009. Discussion period open until June 1, 2010; separate discussions must be submitted for individual papers. This paper is part of the ***Journal of Computing in Civil Engineering***, Vol. 24, No. 1, January 1, 2010. ©ASCE, ISSN 0887-3801/2010/1-95–107/$25.00.

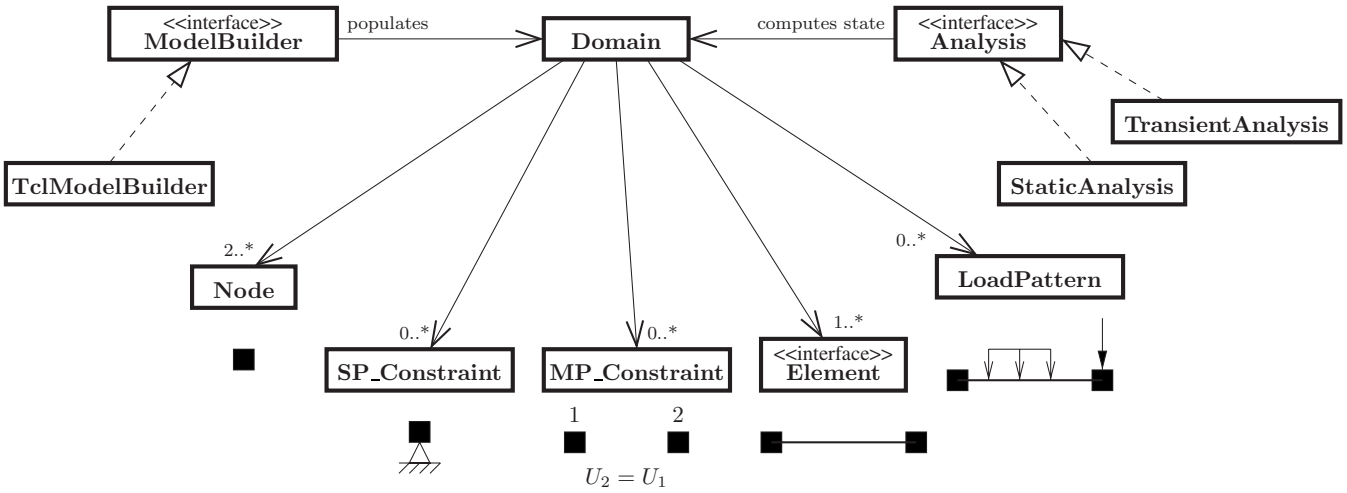


Fig. 1. Class diagram of high-level domain, analysis, and model building classes in the OpenSees framework using the unified modeling language notation

structural analysis and finite-element methods. The encapsulation of data and methods allows object-oriented programs more flexibility and extensibility than equivalent procedure-oriented programs (Rumbaugh et al. 1991; Booch 1994; Sommerville 1995), which can be exploited in engineering software development (Fenves 1990; Baugh and Rehak 1992). A bibliographic listing of object-oriented finite-element implementations between 1990 and 2003 is given by Mackerle (2004). Early works (Forde et al. 1990; Miller 1991; Mackie 1992) demonstrated that object-oriented structural analysis software has shorter development times and is easier to maintain and extend than procedural software. The main drawback to object-oriented software is the computational expense of dynamic memory management, which can account for up to 30% of program execution time (Chang et al. 2001), and random utilization of the memory heap which can cause excessive page faulting in larger programs. This expense can be mitigated by effective programming techniques such as passing references to objects to avoid the dynamic allocation of temporary objects, which is an important consideration for programs written in C++ (Meyers 1997). With effective memory management, the increase in computation time for object-oriented finite analysis over procedural implementations ranges from 10 to 15% (Dubois-Pelerin and Zimmermann 1993; Rucki and Miller 1996).

Recent work to advance research in performance-based earthquake engineering has been organized around the object-oriented software framework OpenSees for structural and geotechnical simulation applications (McKenna et al. 2000). A software framework is a set of classes that a developer can combine and reuse to create an application. The framework defines the abstract classes and provides many of the concrete classes that implement specific functionality for an application space. The abstract classes define a common interface for all users of the class, e.g., an abstract Element class defines methods to compute and return its resisting forces and tangent stiffness. This set of methods is often referred to as an "abstract interface." The concrete classes provide the implementation of the methods declared in the abstract class, or if a method has been implemented in the abstract class, the concrete class can override the method by providing its own implementation.

Developers add functionality by implementing new subclasses or by creating new interfaces that derive or combine behavior from existing components in the framework. The OpenSees framework has classes for representing finite-element models and enabling the solution of the governing equations (McKenna 1997). Fig. 1 shows the high-level classes using the graphical Unified Modeling Language notation (Booch et al. 1998), a standard for expressing object-oriented designs. Central to the framework is the Domain class, which encapsulates the finite-element model, an aggregation of element, node, load pattern, and single- and multipoint constraint objects, whose state determines the structural response.

Although aggregation in the form of the Domain class is a basic example of object composition, the design of a framework for solving finite-element equations to meet the goals of flexibility, extensibility, and portability is more challenging. The first section of the paper presents a finite-element analysis as a composition of loosely coupled algorithms. The subsequent sections are organized around three fundamental concepts in finite-element analysis: representing, forming, and solving the governing equations. Loose coupling is maintained between the software representing these concepts by using design patterns, as demonstrated through the use of the sequence diagrams for assembly and solution of the governing equations. The software design patterns are presented in the context of the OpenSees framework where they have been extensively tested and used by researchers and developers (Miller et al. 2003; Jeremić et al. 2004; Haukaas and Der Kiureghian 2007). The concepts and the specific aspects of the design can be applied to any other finite-element software.

Fundamental Components of a Finite-Element Analysis

The finite-element method discretizes the governing partial differential equations of equilibrium, kinematics, and constitution for a structural problem into a system of nonlinear ordinary differential equations (ODEs) Zienkiewicz and Taylor 2005). The ODEs need to be solved by a time stepping procedure, which is the computationally intensive phase. At a high level, the major steps in a simulation are modeling (including the finite-element

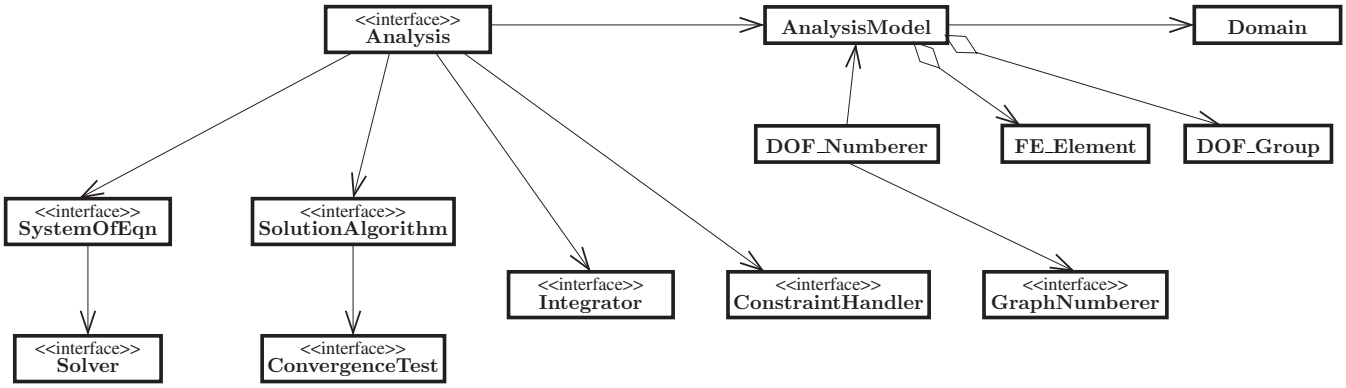


Fig. 2. Analysis class as composition of objects that represent the fundamental components in the solution of finite-element equations

discretization), solving the governing equations, and response interpretation.

The most important operation during the solution phase is to compute the new state of a domain caused by an applied load or the addition or removal of elements. A simple object-oriented approach is to provide a method for the Domain class to implement an analysis procedure to compute its state. The problem with giving the domain responsibility to analyze itself is that there are many types of solution procedures depending on whether the problem is time-dependent or time-independent, whether the equations are assembled or handled locally at nodes, whether a system of equations is solved iteratively or directly, how constraints are included in the governing equations, and other characteristics of nonlinear finite-element analysis procedures.

Uncoupling the analysis procedure from the representation of a domain leads to the need for an analysis class that is separate and distinct from the domain (Rucki and Miller 1996). A common approach to defining the solution procedures is to encapsulate several components of an analysis in a single class in which the programmer creates the necessary data structures to solve the governing equations (Archer et al. 1999). The disadvantage of this approach is that a new analysis class would have to be developed for each combination of analysis components, e.g., time integration method, nonlinear solution algorithm, and constraint handling method. If the solution procedure is changed during the simulation, such as switching from Newton–Raphson to Modified Newton near local extrema of the response, an entirely new analysis object would need to be instantiated. This may require significant overhead if the data structures need to be copied or reorganized for the new solution procedure. Class inheritance may be used to specialize analysis procedures for the many solution combinations; however, this results in a flat hierarchy where many classes have similar implementations, making the system difficult to maintain. It is recognized in object-oriented design and programming that, as a means of extending and reusing code, class inheritance is best used when inheriting an interface of specified operations rather than for inheriting implementations of the operations (Rumbaugh et al. 1991; Booch 1994).

To overcome the problems with class inheritance for solving finite-element equations, a flexible and extensible approach of object composition is used to construct an analysis procedure. In this architecture, illustrated in Fig. 2, an Analysis object is a composition of objects from other classes, each of which is responsible for performing a fundamental operation in determining the state of the finite-element model. This state determination is accomplished by the interaction between the objects in the composition. The Analysis class is abstract and has concrete subclasses that implement a range of analysis procedures, such as for static and transient nonlinear structural analysis. Upon construction, the Analysis object establishes the associations between the objects in its composition, verifies they are interoperable for the analysis procedure, and invokes initialization operations requesting the objects to allocate private data based on the model size. In addition, methods in the Analysis class allow the objects in the aggregation to be changed by the user at any instance in time.

The most important method for the Analysis class is analyze(), which advances the state of the domain for one or more load steps by invoking operations on the objects in its composition entirely through abstract classes rather than the concrete classes with specific implementations. As illustrated in Fig. 3, the implementation of the analyze() method for a static analysis loops over a specified number of steps, forming and solving the equations then committing the solution after the criterion for convergence is satisfied. The only difference in the code for StaticAnalysis and a TransientAnalysis is that the integrator is supplied a time increment for the latter.

A fundamental contribution of this architecture is the use of composition to establish the relationships between the finite-element analysis components to solve the governing equations. Object composition provides greater extensibility and flexibility than is possible when using class inheritance for defining specific finite-element analysis procedures. From a software engineering perspective, this loose coupling of the components in an analysis allows researchers and developers to focus on specific aspects of a finite-element solution procedure with minimal consideration of other aspects. From a user perspective, an analysis can be defined (or redefined) at runtime by providing different combinations of the objects that make up the Analysis composition. For example, in a blast analysis, the first few time steps could be performed using the explicit Central Difference method with a diagonal solver, which could then be switched to an implicit Newmark method and a sparse direct solver for subsequent larger time steps. This is accomplished by the user creating the appropriate objects and passing them to the Analysis object, and it is a process that can be automated via scriptable finite-element analysis where characteristics of the response determine when to switch solution procedures.

There are two important issues to consider when designing classes to perform an analysis. The first is defining the communication between the analysis objects and the domain objects, and

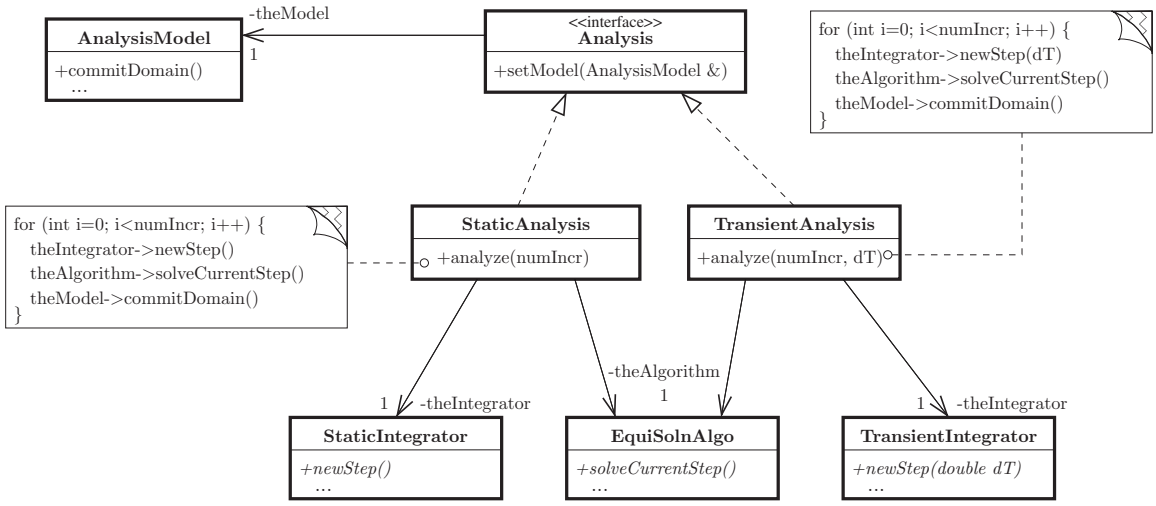


Fig. 3. Class diagram for the analysis showing the analyze() method for static and transient analysis

the second is organizing the analysis objects to represent a wide range of solution methods. As described in the following section, software design patterns provide useful approaches to address these issues.

Software Design Patterns

Design patterns emerged from Alexander (1977), who developed a system to generate modular architectural designs that were adaptable to a wide range of needs. Beck and Cunningham (1987) were the first to utilize design patterns in software development, and patterns have since been applied to several engineering problems (Peckham and MacKellar 2001; Goel and Bhatta 2004; Chen and Adomaitis 2006). Software design patterns abstract the essential relationships between classes, involving both data and operations, that can be adapted to many different situations. A pattern includes the class definitions, relationships, and interactions for a generic problem from which specific implementations can be produced with the desired functionality as well as the flexibility and extensibility offered by the pattern. Gamma et al. (1995) provided an influential catalog of software design patterns that have appeared in many object-oriented designs. By using the recurring mathematical structure in the finite-element method, developers can adapt design patterns to create flexible and extensible finite-element software.

One of the most important software design patterns for computational software is *Strategy* (Gamma et al. 1995), where one of many interchangeable algorithms can be used independently of an application. By encapsulating the algorithms in separate classes, this pattern avoids the code duplication and flat class hierarchy that would result from implementing nearly identical classes that only differ by their strategy. This pattern has appeared in nearly all object-oriented finite-element implementations, e.g., where an element uses interchangeable constitutive models (Scott et al. 2008). In this work, whose focus is nonlinear solution methods, the software patterns described in the following sections are applied to the representation, formation, and solution of governing equations in the finite-element method.

In addition to implementing interchangeable numerical algorithms using *Strategy*, the software design of the Analysis class to address the communication and organization of the classes in Fig. 2 is based on the following design patterns:

- *Iterator*—An iterator pattern is used to hide the internal representation of objects in the finite-element model, which is important for uncoupling finite-element formulations from the way they are used in solving the governing equations.
- *Adapter*—An adapter pattern modifies the response of a finite-element object according to the selected constraint handling and time integration methods such that finite-element implementations are blind to their specific contributions of mass, stiffness, and damping in the governing equations.
- *Factory Method*—The factory method pattern allows the correct adapter objects to be added to the analysis model according to the selected constraint handling method.
- *Bridge*—The bridge pattern decouples the implementation of finite elements from time integration and constraint handling methods such that they can inter-operate independently.

The following three sections describe fundamental aspects of a finite-element analysis, each of which is implemented by classes interacting to represent, form, and solve the governing equations.

Composition of Classes for Representing Finite-Element Equations

Before the governing equations can be solved during an analysis, they must be ordered according to the nodal definition and element connectivity. In addition, all declared constraints must be taken into account in the equation ordering. To avoid tight coupling of finite-element implementations and the analysis methods, the AnalysisModel provides a layer of abstraction between objects in the domain and the governing equations. The AnalysisModel is a composition of DOF_Group and FE_Element objects, which allow developers of finite-element classes to not be concerned with the specific operations that form and solve the governing equations.

DOF_Group Class

A DOF_Group object represents and operates on the nodal degrees of freedom (DOFs) in the domain. The default implemen-

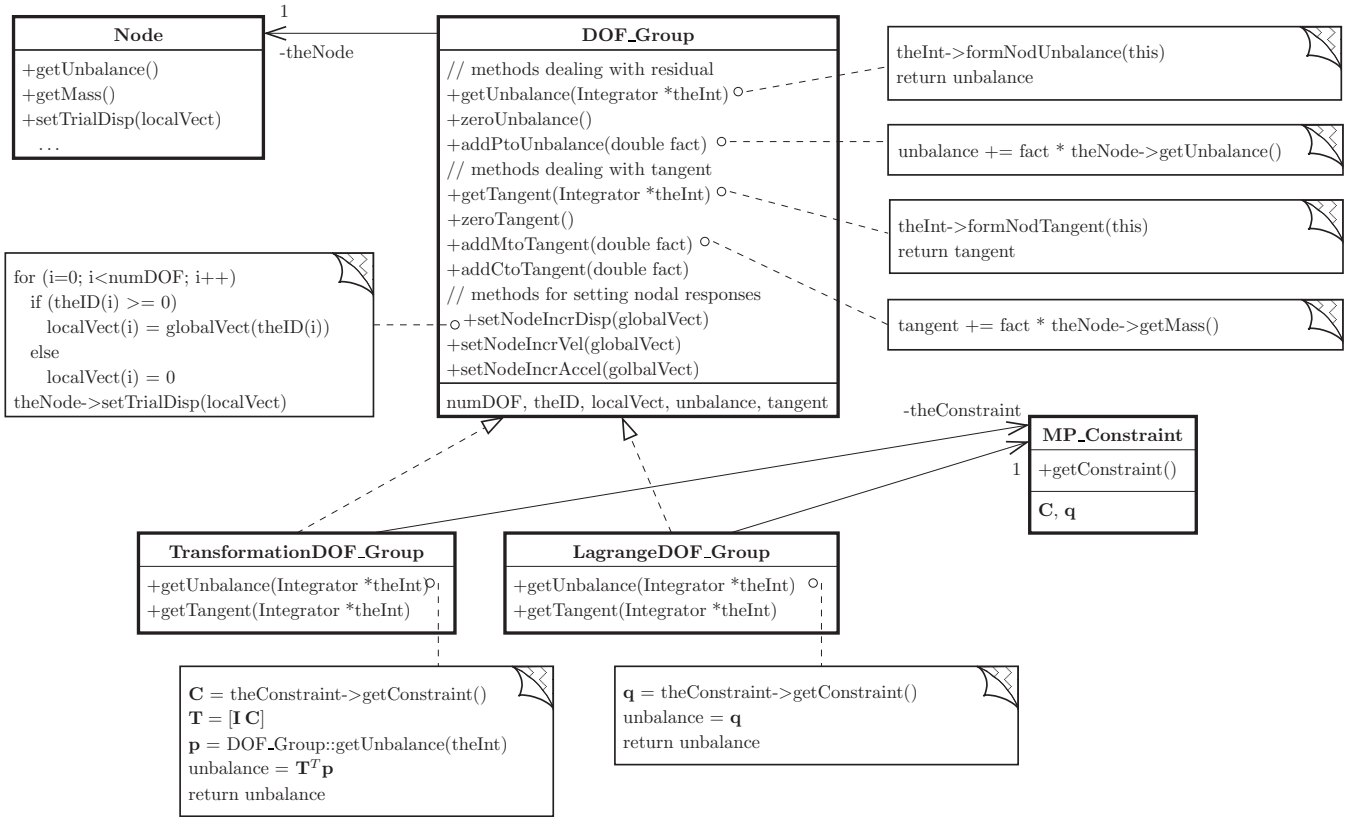


Fig. 4. Class diagram of the DOF_Group class showing default methods to form nodal unbalance and tangent, as well as specific implementations for transformation and Lagrange constraint handlers

tation contains a connectivity vector that gives the equation numbers for the DOFs at a single Node object in the domain, as shown in Fig. 4. Methods are provided to relate response quantities between the nodal and structural DOFs. Subclasses of DOF_Group override this default behavior for the case where the system of equations must be modified to handle constraints, such as by transforming nodal response before assembly or by introducing additional equations involving Lagrange multipliers. This design, based on the *Adapter* pattern (Gamma et al. 1995), keeps the implementation of the Node class separate from the implementation of constraint handling procedures, thereby alleviating the need to develop subclasses of Node for every type of constraint handler.

FE_Element Class

The FE_Element class processes finite-element response quantities prior to their assembly in the governing equations. As shown in Fig. 5, the FE_Element class maintains a reference to a single finite-element object in the domain. A FE_Element obtains the element residual vector and stiffness, mass, and damping matrices, and may modify them before assembling their contributions to the tangent stiffness for the connected nodes, whose DOFs and equation numbers are accessed via the DOF_Group object. Similar to DOF_Group, the FE_Element class is based on the *Adapter* pattern. It provides a great deal of flexibility in how the finite-element formulation is incorporated into the governing equations. Considering the software life cycle, this flexibility outweighs the cost of additional memory and method calls required to adapt the finite-element response.

Since the procedure for forming the tangent depends on the integration method and the resulting contributions of mass, damping, and stiffness, an FE_Element object delegates to the integrator the task of forming the tangent for the element by providing callback functions such as addKtoTang(). Similar callback operations form the element residual vector. Other FE_Element operations add terms to the tangent matrix for the element accounting for damping and mass. Subclasses of FE_Element implement procedures for modifying the element residual and stiffness to account for constraints in the finite-element model.

AnalysisModel Class

The AnalysisModel is a container class that stores and provides access to the FE_Element and DOF_Group objects that have been created for an analysis. The methods getFEs() and getDOFs() use the *Iterator* pattern (Gamma et al. 1995) for sequential access to the FE_Element and DOF_Group objects in the model without exposing their internal representation in the analysis model. This approach is advantageous for assembling contributions from elements that do not reside in main memory, such as for parallel processing or hybrid simulation (Takahashi and Fenves 2006). Other methods in the interface query and modify the state of the domain. A reference to the Domain is maintained such that the AnalysisModel can commit the state of all components in the domain upon convergence of the numerical solution at each time step. The AnalysisModel is also responsible for building and returning a graph of the connectivity of all DOFs that have been added to the model.

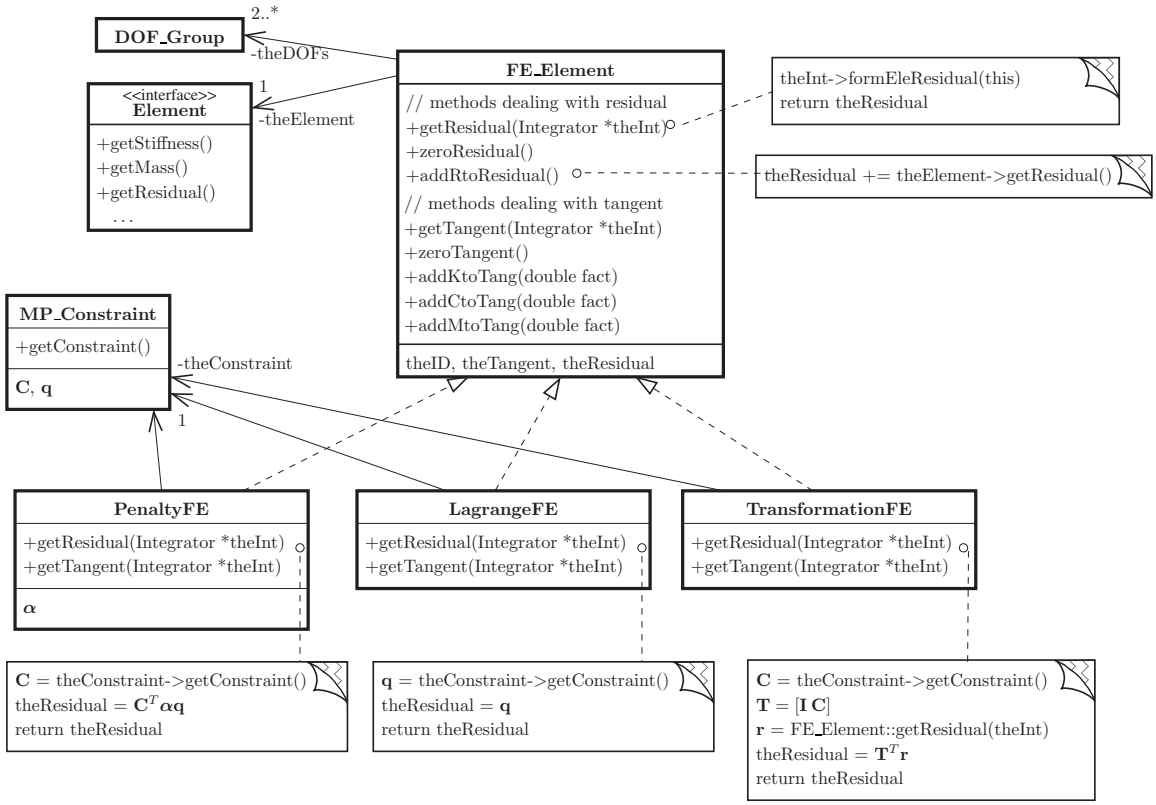


Fig. 5. Class diagram of the FE_Element class showing implementations of the getResidual() method for different constraint handling techniques

In addition to storing the representation of the finite-element equations, the AnalysisModel uses a DOF_Numberer to number the equations in order to take advantage of the sparsity of the stiffness matrix. As shown in Fig. 6, the DOF_Numberer obtains the connectivity graph from the AnalysisModel, then uses a strategy, e.g., minimum degree (Tinney and Walker 1967) and reverse Cuthill-McKee (Cuthill and McKee 1969), encapsulated by subclasses of GraphNumberer in order to assign equation numbers for each DOF_Group. The DOF_Numberer then iterates over all FE_Element objects and determines their DOF mapping of connected nodes based on the numbering assigned to the DOF_Group objects.

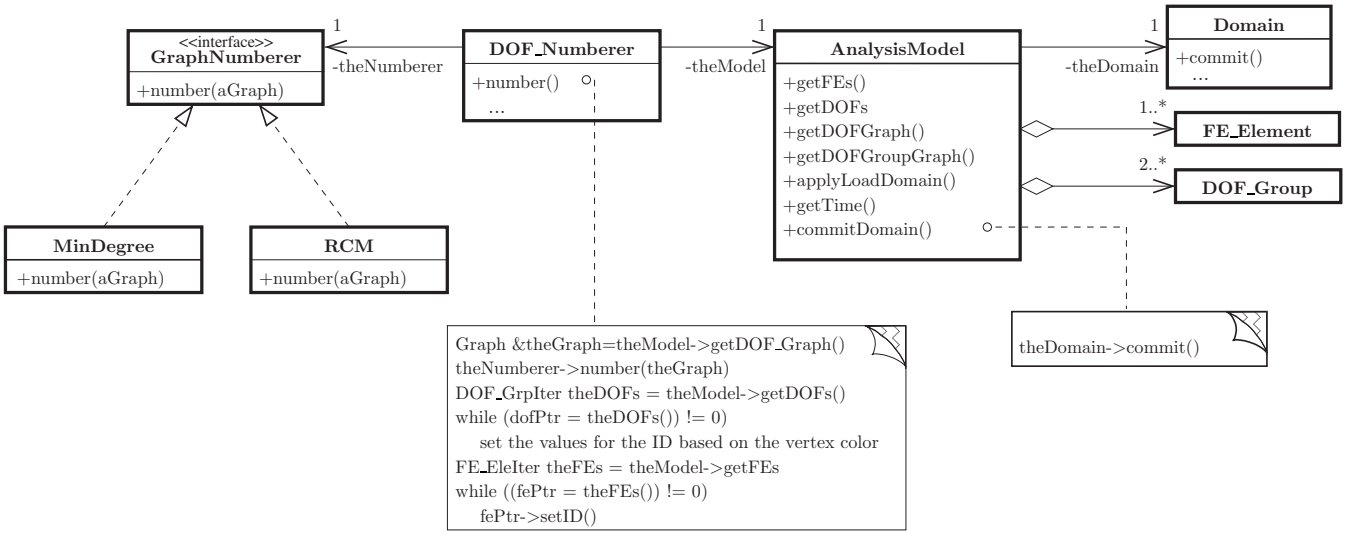


Fig. 6. Implementation of the AnalysisModel class as a container class for FE_Element and DOF_Group objects and the DOF_Numberer class using a graph numbering strategy

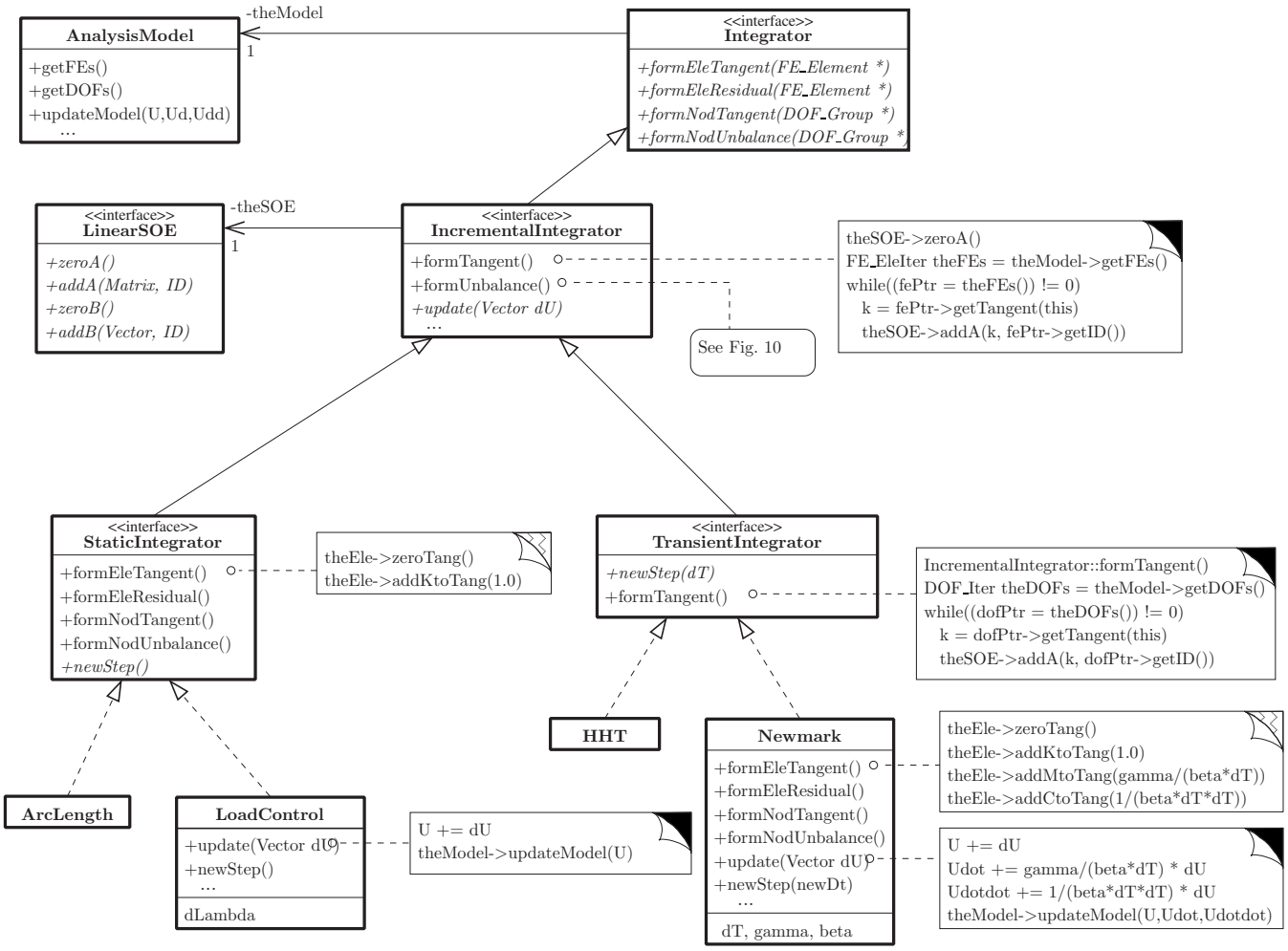


Fig. 7. Class diagram for the IncrementalIntegrator showing key implementations for equation assembly

Pattern-Based Implementation of Forming Finite-Element Equations

With mechanisms in place to map element and nodal response to the governing equations, classes are defined in this section to assemble the equations and to process single- and multipoint constraints. The *Bridge* and *Factory Method* patterns make the equation assembly flexible and extensible such that the framework can accommodate a wide range of time integration and constraint handling methods.

Integrator Class

The Integrator class is the main link between the numerical solution procedures and the state of the finite-element model. It is responsible for assembling the governing equations, as well as recovering nodal response quantities and updating the state of the finite-element model after obtaining the equation solution from the analysis composition.

As shown in Fig. 7, the Integrator class is abstract with methods to form the residual and tangent of FE_Element and DOF_Group objects. Subclasses of Integrator extend the interface with additional operations for the particular type of equations to be formed during an analysis. For example, the IncrementalIntegrator class includes methods to form the tangent and residual during a nonlinear analysis. The tangent matrix of the residual equilibrium equations is formed by first zeroing the matrix, then iterating over all FE_Element objects in the analysis model and assembling their contributions. The getTangent() method invoked on the FE_Element objects initiates a sequence of callback functions to assemble the element tangent matrix in accordance with the parameters for the time integration method. A similar system of callback methods is used to assemble nodal contributions via the DOF_Group objects in the analysis model.

The implementation of the callback methods to form the element and nodal contributions to the tangent and residual of the finite-element model is delegated to subclasses of IncrementalIntegrator. In the case of a static analysis, the element residual vector and tangent matrix do not need to be modified prior to assembly. This functionality is implemented in the StaticIntegrator class while the calculation of the load increment is deferred to its subclasses, e.g., LoadControl and ArcLength. On the other hand, the modification of the element residual and tangent is deferred to subclasses of TransientIntegrator for dynamic analyses that require a time integration method such as the implicit Newmark, Wilson-θ, Hilber-Hughes-Taylor (HHT)-α, and Collocation methods, and the explicit Central Difference and explicit variations of the Newmark and HHT-α methods. A TransientIntegrator object uses the nodal displacement vector to update the finite-element model consistent with the aforementioned

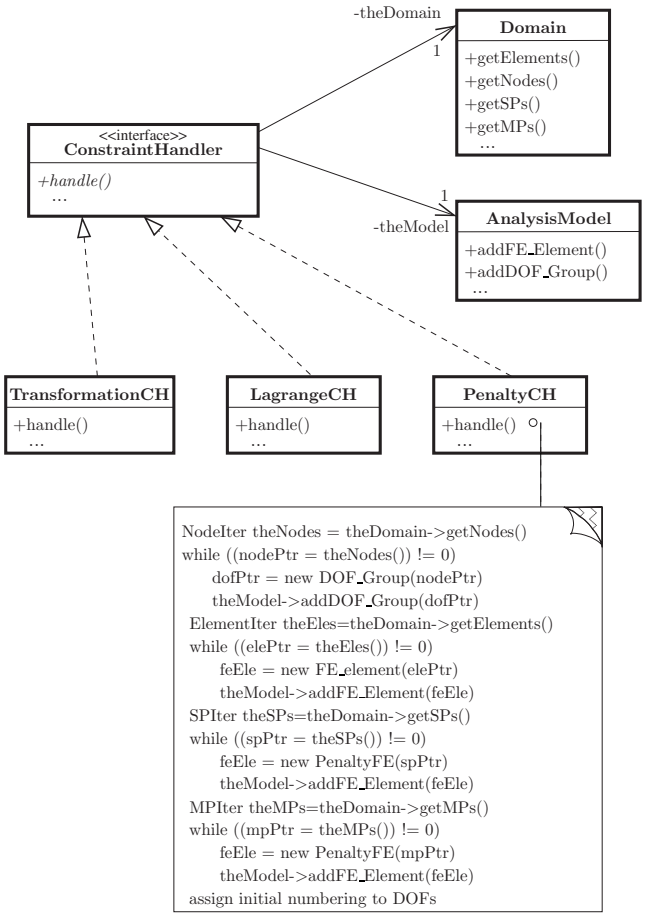


Fig. 8. Class diagram for the ConstraintHandler where subclasses are responsible for populating the AnalysisModel with the adapter objects that correspond to the constraint handling method

time integration methods. Further details of the architecture for equation assembly are given in a subsequent section using sequence diagrams.

The communication between integrator, element, and node objects is based on the *Bridge* design pattern (Gamma et al. 1995), where the implementations of these objects are completely uncoupled, which is an important consideration when extending the framework to problems with different types of nodal DOFs. Furthermore, it allows the integrator to be changed at runtime without affecting other components of the analysis or the objects in the finite-element model.

Constraint Handler Class

Depending on how constraints are included in the analysis, modifications to the system of equations and/or to the element contributions during equation assembly may be necessary. While the specific constraint handling methods can be implemented using the *Strategy* design pattern shown in Fig. 8, the use of an additional pattern, *Factory Method* (Gamma et al. 1995), provides the desirable loose coupling between the constraint handling method and the nodes and elements in the analysis model. To this end, the ConstraintHandler class declares functionality to return DOF_Group and FE_Element objects of the type corresponding to the constraint handling method used in the analysis. Subclasses of ConstraintHandler instantiate and return to the AnalysisModel the FE_Element and DOF_Group objects that will enforce constraint equations in the element and nodal response using the selected constraint handling method.

The design decision to construct DOF_Group and FE_Element objects within subclasses of ConstraintHandler keeps the implementation of the AnalysisModel blind to the constraint handling method and the associated adapter objects it contains. The objects created by subclasses of ConstraintHandler are completely decoupled from implementations of the Integrator class, as well as from the element and node implementations in the domain.

Strategies for Solving Finite-Element Equations

Once the governing equations have been formed during an analysis time step, several algorithms can be used to obtain a solution. For the solution to residual equilibrium equations in a nonlinear analysis, classes encapsulating root-finding algorithms and linear equation solvers work in concert to determine the primary nodal unknowns of the system. The implementation of these classes follows the *Strategy* design pattern.

Solution Algorithm Class

A solution algorithm object is responsible for specifying the steps to solve the equations at the current time step. The SolutionAlgorithm class is abstract; each subclass provides a specific implementation of an algorithm. For the incremental solution of equilibrium equations in static and transient analysis, the subclass is EquiSolnAlgorithm. This class implements the solution to determine the root of the residual equilibrium equation

$$\mathbf{R}[\mathbf{U}(t)] = \mathbf{P}_f(t) - \mathbf{P}_r[\mathbf{U}(t)] = \mathbf{0} \tag{1}$$

where $\mathbf{P}_f$=time-dependent nodal load vector and $\mathbf{P}_r$=resisting force vector which is a nonlinear function of the nodal displacements $\mathbf{U}$, and is assembled from element contributions. An iterative approach is taken to find the root of Eq. (1) at time step k

$$\mathbf{U}_k^{j+1} = \mathbf{U}_k^j + \Delta\mathbf{U}_k^{j+1} \tag{2}$$

where j counts the iterations for the time step.

Subclasses of EquiSolnAlgorithm implement the solveCurrentStep() in order to solve Eq. (1), e.g., using the Newton–Raphson algorithm for implicit methods or a linear algorithm (one solve with no subsequent iteration) for explicit methods. An EquiSolnAlgorithm object defines the strategy for StaticAnalysis and TransientAnalysis classes to solve the equilibrium equations. The encapsulation of the solution algorithm in a separate class allows the algorithm to be changed during an analysis with only the data associated with the algorithm needing to be reallocated.

SystemOfEqn and Solver Classes

The solution to a linearized system of equations is an essential step to finding the solution of Eq. (1) by iterative root-finding algorithms. For this class of algorithms, the displacement increment $\Delta\mathbf{U}_k^{j+1}$ of Eq. (2) is obtained from the solution to the following linear system of equations:

$$\mathbf{K}\Delta\mathbf{U}_k^{j+1} = \mathbf{R}_k^j \tag{3}$$

where $\mathbf{K}$=stiffness matrix of the structure, which may be assembled from elements prior to or during the equation solution depending on the solver.

There are many direct and iterative equation solvers that can be used to solve Eq. (3). However, encapsulating in a single class

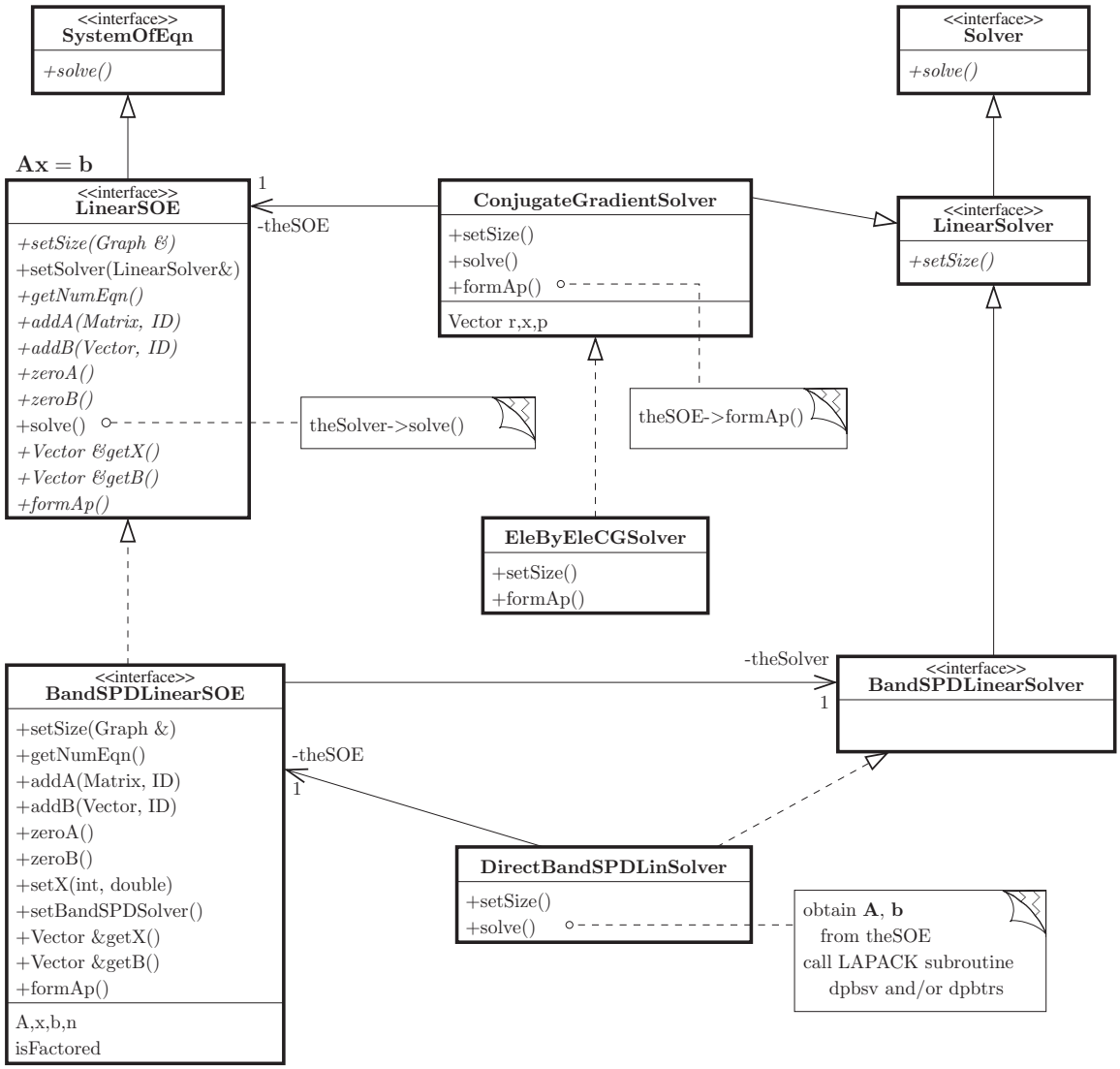


Fig. 9. Class diagram of the LinearSOE and LinearSolver showing implementations of a Banded SPD direct solver and the conjugate gradient iterative solver

the equation storage and the operations to solve the system can lead to code duplication of the matrix storage scheme when implementing new linear equation solvers. To avoid duplication and improve flexibility in solving linear equations, the matrix representation is decoupled from the equation solver using two abstract interfaces: SystemOfEqn and Solver. Subclasses are provided for storing and solving particular types of equations, e.g., the LinearSOE and LinearSolver classes shown in Fig. 9 declare interfaces for equations of the form **Ax=b**.

Subclasses of LinearSOE implement a storage scheme for a matrix whose topology may be exploited by a direct solver and also serve as a "black box" for iterative equation solvers. Direct and iterative equation solvers are encapsulated by subclasses of LinearSolver. As an example, the BandSPDLinearSOE class shown in Fig. 9 implements a banded, symmetric positive definite (SPD) storage of the stiffness matrix. A variety of LinearSolver implementations operate on the banded SPD system, including the DirectBandSPDLinSolver, which uses the LAPACK numerical library (Anderson et al. 1995), and the ConjugateGradientSolver, which asks the LinearSOE to multiply its matrix by an arbitrary vector. Additional subclasses of LinearSOE and LinearSolver classes implement storage schemes and direct solvers for other matrix topologies. The separation of the system from the solver provides a many-to-one relationship between solvers and equation storage, so that many direct and iterative sparse equation solvers can operate on a single storage format (Barrett et al. 1994).

Sequence Diagrams for Analysis Operations

The previous sections presented a static view of the classes contributing to the Analysis composition and several of the key operations. To describe how the Analysis objects interoperate to form and solve the governing equations, this section presents sequence diagrams of analysis steps. Sequence diagrams provide an overview of important aspects of software by showing runtime objects and messages passed between them (Booch et al. 1998). The sequence diagrams presented in this section give a complete picture of the loose coupling between objects that assemble the governing equations and the objects that contribute to the equations.

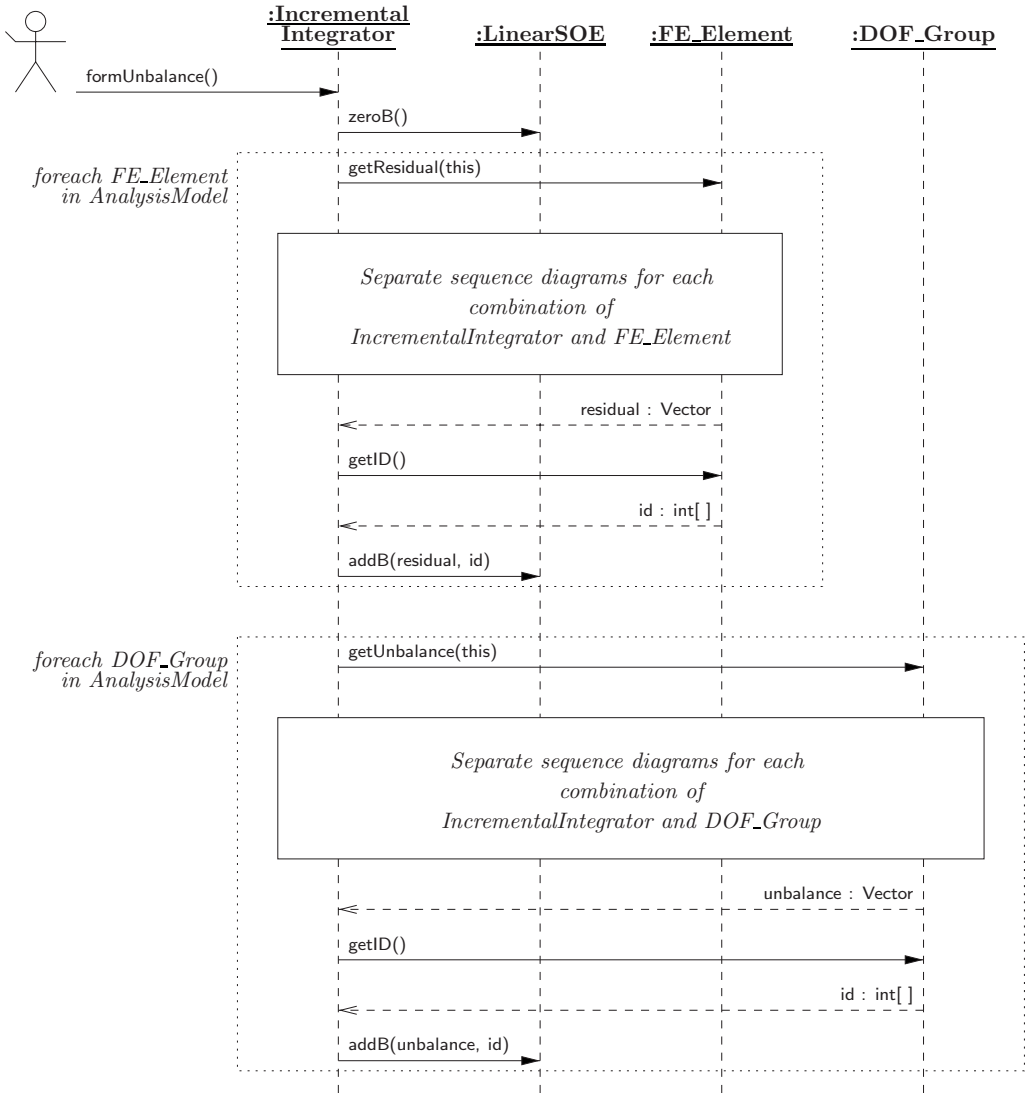


Fig. 10. Sequence diagram for the formUnbalance() method of the IncrementalIntegrator class

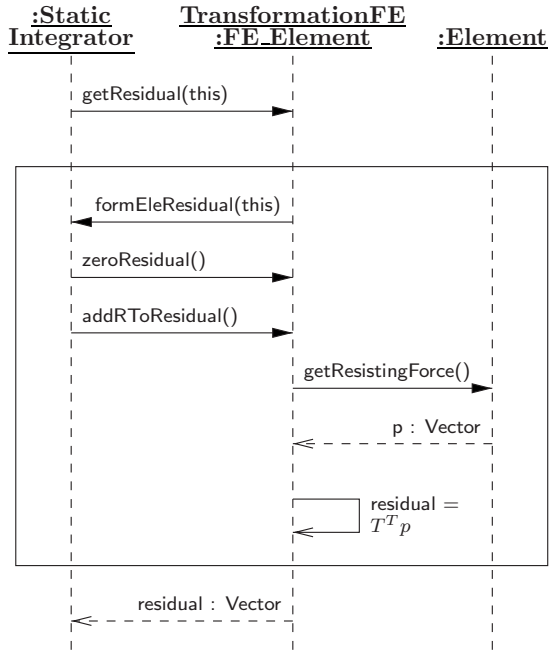
A key step in the numerical solution of finite-element equations is the assembly of the residual vector in Eq. (3). The methods invoked during the formUnbalance() operation of the IncrementalIntegrator class are shown in Fig. 10. After zeroing the right-hand side of the LinearSOE object, the IncrementalIntegrator iterates over the FE_Element objects in the analysis model to form the residual vector. The getResidual() method call to the FE_Element interface initiates a sequence of operations to form the element residual considering the time integration method and constraint handler in the analysis composition. One such combination is shown in Fig. 11(a) for static time integration with a transformation constraint handling method. In this case the element residual is transformed prior to the StaticIntegrator object assembling its contribution to the governing equations. Inertial effects are obtained from the element objects for the case of transient Newmark time integration shown in the sequence diagram of Fig. 11(b). Returning to Fig. 10 after all FE_Element object contributions have been assembled, subsequent iteration over all DOF_Group objects adds unbalanced nodal loads to the governing equations. Sequences of method calls similar to those shown in Fig. 11 assemble the nodal response for different combinations of time integration and constraint handling methods.

To solve the governing nonlinear equations, the EquiSolnAlgorithm class obtains the finite-element residual vector by calling the formUnbalance() method on the IncrementalIntegrator. The sequence diagram of the Newton–Raphson implementation of the solveCurrentStep() method is shown in Fig. 12. The standard steps of forming the residual vector and tangent stiffness matrix are executed by passing messages to the IncrementalIntegrator object. After forming the linearized system of equations, the

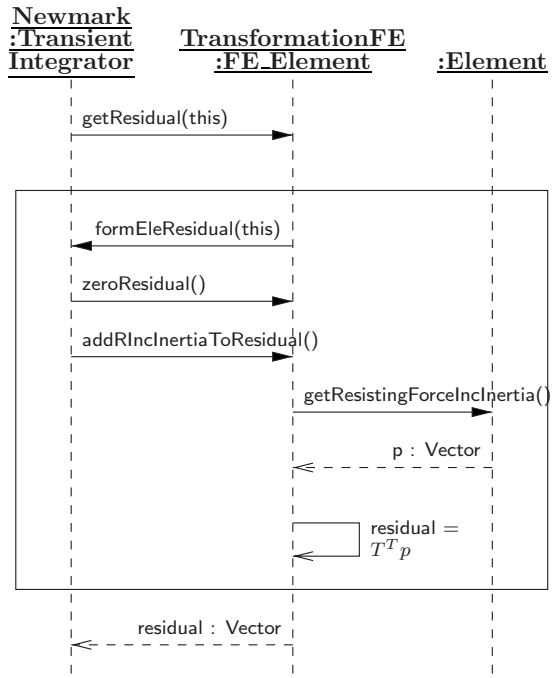
algorithm obtains the solution for the displacement increment from the LinearSOE object, then passes the solution to the IncrementalIntegrator in order to recover all nodal response quantities and update the finite-element model. Other solution algorithms are implemented by making different sequences of abstract calls to the IncrementalIntegrator and LinearSOE objects, as shown in the following section for line search algorithms.

Example of Software Extensibility

The implementation of line search algorithms in conjunction with the Newton–Raphson solution algorithm provides a useful example of the extensibility of the nonlinear finite-element analysis



(a) Static integration



(b) Transient time integration

Fig. 11. Sequence diagrams showing the interaction of incremental integrators and FE_Elements for: (a) Static integrator forming the residual of transformation FE_Element objects; (b) Newmark transient integrator forming the residual including inertial effects of transformation FE_Element objects.

framework afforded by design patterns. As described earlier, the equilibrium solution algorithms follows the *Strategy* pattern in which an Analysis object uses one of many interchangeable implementations. This pattern can be applied further to implement line search algorithms that improve the convergence of the Newton–Raphson algorithm (Crisfield 1991; Bathe 1996). As shown in Fig. 13, the NewtonLineSearch class invokes the

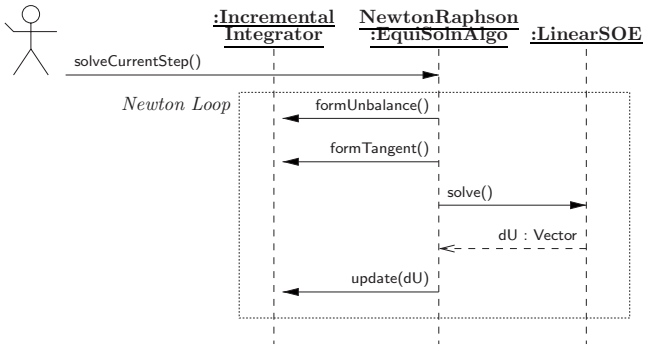


Fig. 12. Sequence diagram for one time step using the Newton–Raphson solution algorithm

search() method via the LineSearch interface. Subclasses of LineSearch implement specific line search techniques, such as bisection, regula falsi, and secant. A sequence diagram for the solveCurrentStep() method of the NewtonLineSearch class is shown in Fig. 14, where additional method calls are made to the LineSearch object during the search for a solution. The inclusion of line search algorithms was not considered during the initial framework design; however, the encapsulation of solution algorithms in classes separate from the analysis provides the necessary flexibility to incorporate line search algorithms without affecting other classes in the framework.

Conclusions

This paper has defined the key aspects of implementing an object-oriented framework for representing, forming, and solving nonlinear finite-element equations using object composition as the primary means of achieving flexibility, extensibility, and portability. The framework design based on object composition takes advantage of software design patterns to define communication between objects in the analysis. The *Strategy* pattern defines an interface for interchangeable algorithms of all major steps in nonlinear finite-element analysis, while a layer of abstraction between the finite-element model and the analysis components is provided by the *Iterator* and *Adapter* patterns. Finite-element analysis implementations are decoupled from the equation assembly through the *Factory Method* and *Bridge* patterns. Because of the flexibility offered by these and other design patterns, applications such as structural reliability, adaptive mesh refinement, meshless finite-element methods, hybrid simulation, and contact problems can be built using the framework. Future areas of research will focus on the computational performance of design pattern based implementations of nonlinear finite-element analysis, particularly for parallel computation with explicit and mixed implicit-explicit methods, as well as compiler support for memory management techniques that improve the computational performance of object-oriented software.

Acknowledgments

This work and the development of OpenSees has been supported by the Pacific Earthquake Engineering Research Center under Grant No. EEC-9701568 from the National Science Foundation. The source code and full application program interface (API) of the classes in the OpenSees framework, including the

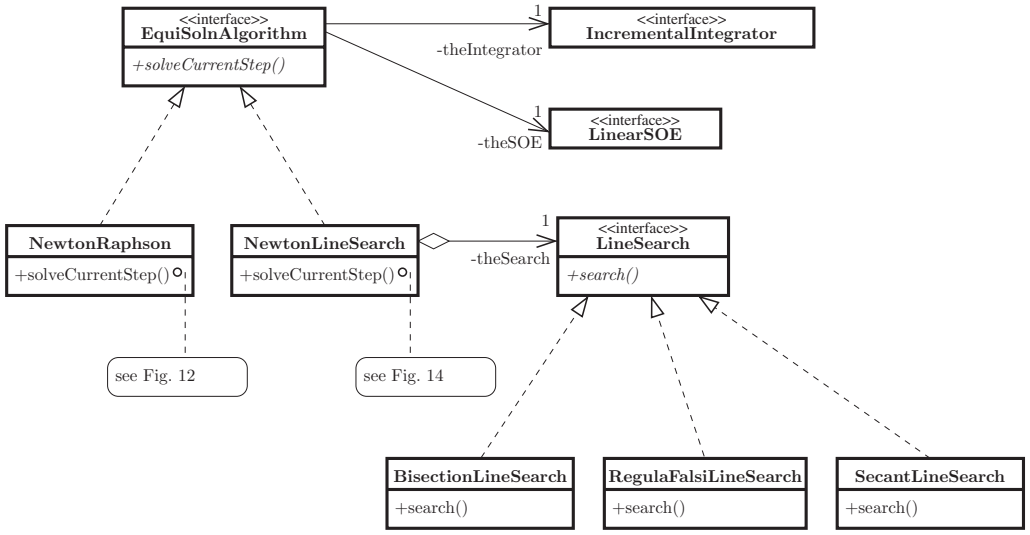


Fig. 13. Class diagram of the EquiSolnAlgorithm class showing subclasses for standard Newton–Raphson and Newton using a line search strategy

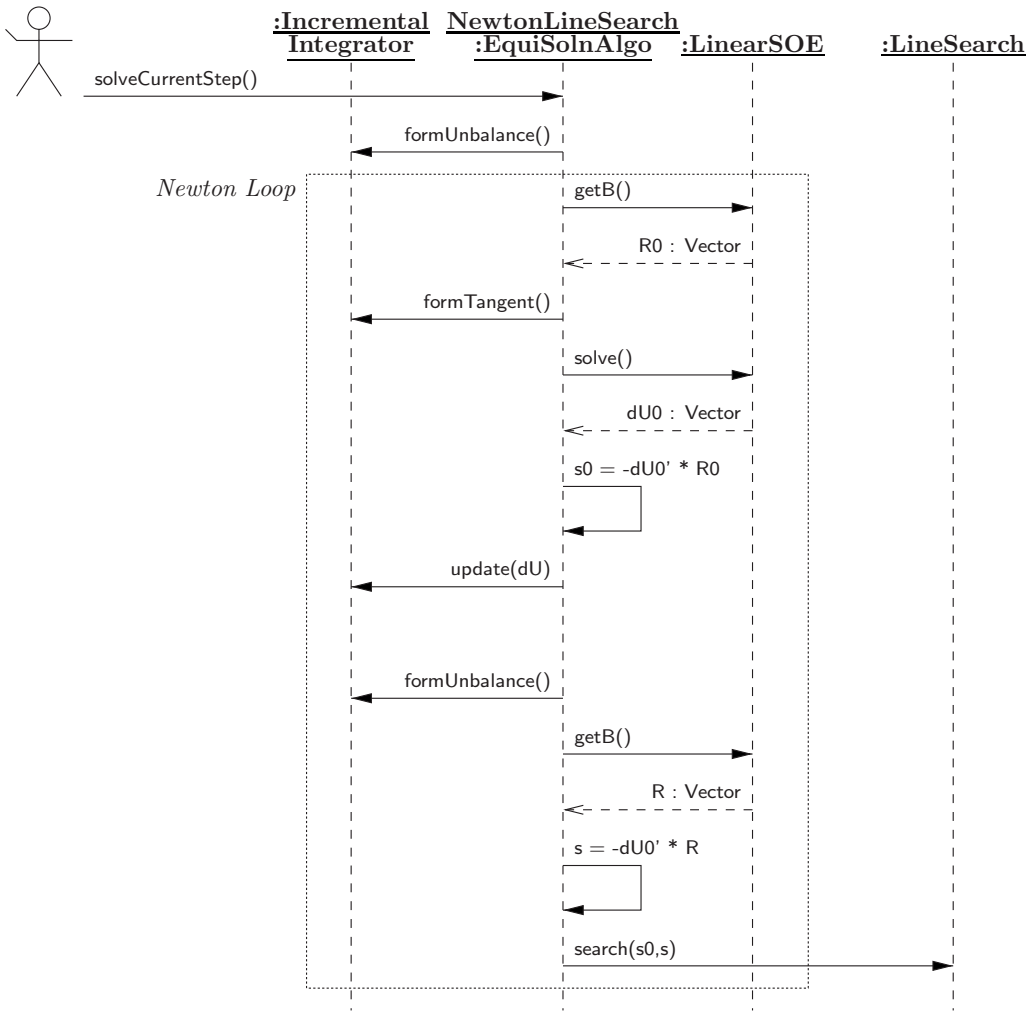


Fig. 14. Sequence diagram for one time step using the Newton–Raphson algorithm with line search

analysis classes defined in this paper, are available at http://opensees.berkeley.edu. The writers thank Dr. Ed Love for implementing line search algorithms within the OpenSees framework.

References

Alexander, C. (1977). *A pattern language: Towns, buildings, construction*, Oxford University Press, New York.

Anderson, E., et al. (1995). *LAPACK users guide*, 2nd Ed., SIAM, Philadelphia, Pa.

Archer, G. C., Fenves, G., and Thewalt, C. (1999). "A new object-oriented finite-element analysis architecture." *Comput. Struct.*, 70(1), 63–75.

Barrett, R., et al. (1994). *Templates for the solution of linear systems*, SIAM,Philadelphia.

Bathe, K. J. (1996). *Finite-element procedures*, Prentice-Hall, Upper Saddle River, N.J.

Baugh, J. W., and Rehak, D. R. (1992). "Data abstraction in engineering software development." *J. Comput. Civ. Eng.*, 6(3), 282–301.

Beck, K., and Cunningham, W. (1987). "Using pattern languages for object-oriented programs." *Rep. No. CR-87-43*, Tektronix, *OOPSLA '87 Workshop on Specification and Design for Object-Oriented Programming*.

Booch, G. (1994). *Object-oriented analysis and design with applications*, Addison-Wesley, Reading, Mass.

Booch, G., Rumbaugh, J., and Jacobson, I. (1998). *The unified modeling language user's guide*, Addison-Wesley, Reading, Mass.

Chang, J. M., Lee, W. H., and Witawas, S. (2001). "A study of the allocation behavior of C++ programs." *J. Syst. Softw.*, 57(2), 107–118.

Chen, J., and Adomaitis, R. A. (2006). "An object-oriented framework for modular chemical process simulation with semiconductor processing applications." *Comput. Chem. Eng.*, 30(9), 1354–1380.

Crisfield, M. A. (1991). *Non-linear finite element analysis of solids and structures*, Vol. 1, Wiley, New York.

Cuthill, E., and McKee, J. (1969). "Reducing the bandwidth of sparse symmetric matrices." *1969 24th National Conf.*, ACM Press, New York, N.Y., 157–172.

Dubois-Pelerin, Y., and Zimmermann, T. (1993). "Object-oriented finite-element programming. III: An efficient implementation in C++." *Comput. Methods Appl. Mech. Eng.*, 108, 165–183.

Fenves, G. L. (1990). "Object-oriented programming for engineering software development." *Eng. Comput.*, 6(1), 1–15.

Forde, B. W. R., Foschi, R. O., and Stiemer, S. F. (1990). "Object-oriented finite-element analysis." *Comput. Struct.*, 34(3), 355–374.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, Reading, Mass.

Goel, A. K., and Bhatta, S. R. (2004). "Use of design patterns in analogy-based design." *Adv. Eng. Inf.*, 18(2), 85–94.

Haukaas, T., and Der Kiureghian, A. (2007). "Methods and object-oriented software for FE reliability and sensitivity analysis with application to a bridge structure." *J. Comput. Civ. Eng.*, 21(3), 151–163.

Hughes, T. J. R. (1987). *The finite-element method*, Prentice-Hall, Englewood Cliffs, N.J.

Jeremić, B., Kunnath, S. K., and Xiong, F. (2004). "Influence of soil-foundation-structure interaction on seismic response of the I-880 viaduct." *Eng. Struct.*, 26, 391–402.

Mackerle, J. (2004). "Object-oriented programming in FEM and BEM: A bibliography (1990–2003)." *Adv. Eng. Software*, 35(6), 325–336.

Mackie, R. I. (1992). "Object-oriented programming of the finite-element method." *Int. J. Numer. Methods Eng.*, 35(2), 425–436.

McKenna, F. (1997). "Object-oriented finite-element programming: Frameworks for analysis, algorithms, and parallel computing." Ph.D. thesis, Univ. of California, Berkeley, Calif.

McKenna, F., Fenves, G. L., and Scott, M. H. (2000). "Open system for earthquake engineering simulation." *Univ. of California, Berkeley, Calif.* ⟨http://opensees.berkeley.edu⟩ (Dec. 6, 2000).

Meyers, S. (1997). *Effective C++*, 2nd Ed., Addison-Wesley, Reading, Mass.

Miller, G. R. (1991). "An object-oriented approach to structural analysis and design." *Comput. Struct.*, 40(1), 75–82.

Miller, G. R., Arduino, P., Jang, J., and Choi, C. (2003). "Localized tensor-based solver for interactive finite-element applications using C++ and java." *Comput. Struct.*, 81, 423–427.

Modak, S., and Sotelino, E. D. (2002). "An object-oriented parallel programming framework for linear and nonlinear transient analysis of structures." *Comput. Struct.*, 80(1), 77–84.

Moehle, J. P., and Deierlein, G. G. (2004). "A framework methodology for performance-based earthquake engineering." *Proc., 13th World Conf. on Earthquake Engineering*, Vancouver, BC, Canada, Paper No. 679.

Peckham, J., and MacKellar, B. (2001). "Generating code for engineering design systems using software patterns." *Artif. Intell. Eng.*, 15(2), 219–226.

Peng, J., Liu, J., Law, K. H., and Elgamal, A. (2004). "ParCYCLIC: Finite-element modelling of earthquake liquefaction response on parallel computers." *Int. J. Numer. Analyt. Meth. Geomech.*, 28(12), 1207–1232.

Rucki, M. D., and Miller, G. R. (1996). "An algorithmic framework for flexible finite-element-based structural modeling." *Comput. Methods Appl. Mech. Eng.*, 136(3–4), 363–384.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented modeling and design*, Prentice-Hall, Englewood Cliffs, N.J.

Scott, M. H., Fenves, G. L., McKenna, F. T., and Filippou, F. C. (2008). "Software patterns for nonlinear beam-column models." *J. Struct. Eng.*, 134(4), 562–571.

Sommerville, I. (1995). *Software engineering*, 5th Ed., Addison-Wesley, Reading, Mass.

Takahashi, Y., and Fenves, G. L. (2006). "Software framework for distributed experimental-computational simulation of structural systems." *Earthquake Eng. Struct. Dyn.*, 35(3), 267–291.

Tinney, W. F., and Walker, J. W. (1967). "Direct solution of sparse network equations by optimally ordered triangular factorization." *Proc. IEEE*, 55, 1801–1809.

Zienkiewicz, O. C., and Taylor, R. L. (2005). *The finite-element method for solid and structural mechanics*, 6th Ed., Butterworth-Heinman, Stoneham, Mass.