Computers and Structures 87 (2009) 952–961

Contents lists available at ScienceDirect

Computers and Structures

journal homepage: www.elsevier.com/locate/compstruc

Using design patterns in object-oriented finite element programming

B.C.P. Heng, R.I. Mackie *

Civil Engineering, School of Engineering, Physics and Mathematics, University of Dundee, Dundee DD1 4HN, United Kingdom

ARTICLE INFO

Article history:
Received 5 October 2006
Accepted 29 April 2008
Available online 11 June 2008

Keywords:
Design patterns
Object-oriented
Finite element method

ABSTRACT

This paper proposes the use of design patterns to capture best practices in object-oriented finite element programming. Five basic design patterns are presented. In *Model-Analysis separation*, analysis-related classes are separated from those related to finite element modelling. *Model-UI separation* separates responsibilities related to the user interface from model classes. *Modular Element* uses object composition to reduce duplication in element type classes while avoiding the problems associated with class inheritance. *Composite Element* lets clients handle substructures and elements uniformly. Decomposing the analysis subsystem as in *Modular Analyzer* increases reuse and flexibility. Alternative solutions to each pattern are also reviewed.

1. Introduction

The first object-oriented (O-O) implementations of the finite element method were put forward more than 15 years ago. Since that time, numerous approaches have been proposed. The design of an O-O finite element program is affected by a number of factors, including software requirements, language features, executing environment, etc. The developer's methodology and viewpoint are also important factors. Varying degrees of object orientation – even procedural design – can be accomplished using an O-O language.

Given such a variety of factors, it is not surprising to find differences in program design. On the other hand, there are similarities too. These similarities are instructive because they reflect consensus among researchers. As this field of research continues to mature, best practices in program design will begin to emerge. It would be useful to capture the key features of these practices in a language-independent and reusable format. Design patterns are a means of achieving this goal.

Software design patterns were popularised by Gamma et al. [1]. A design pattern is the abstraction of a recurring solution to a design problem. It captures the relationships between objects participating in the solution and describes their collaborations. By facilitating reuse of proven solutions, design patterns help to improve software quality and reduce development time. In addition, pattern names form a vocabulary that allows developers to communicate their designs effectively.

Gamma et al. [1] documented 23 general design patterns that have since become popular among O-O developers. Liu et al. [2] and Fenves et al. [3] explicitly used some of these patterns in their finite element systems. However, there are problems with using general design patterns. It still requires much time and effort to identify the specific areas in which these patterns may be used. Furthermore, scientific software developers may not be familiar with them. There is a need therefore to discover and document patterns that are specific to finite element programming.

A documentation of the patterns of O-O finite element software could serve as a common knowledge base for researchers and software developers in this field. It also represents a step towards unifying the different approaches to program design. A common base architecture in turn facilitates collaboration and reuse among development teams.

This paper will use design patterns to identify best practice in object-oriented finite element program design. The next section will describe the methodology used and how design patterns work. The methodology is then applied to five design patterns. The paper closes with conclusions drawn from the work.

Most object-oriented work in finite element analysis has been implemented in C++. However, Java has also been used by some [4], as has C# [5]. The work described here has been developed within a C# context, but is generally applicable to object-oriented programming in any language.

2. Methodology

A three-phase approach was adopted in this work. The first phase involved finding similarities among O-O finite element implementations in the literature. Differences in programming language and presentation format made it difficult to compare program designs. It was necessary therefore to translate designs into a common graphical language. The Unified Modelling Language (UML) [6] – the de facto standard for O-O modelling – was chosen

* Corresponding author. Tel.: +44 1382 384702; fax: +44 1382 384816.
E-mail address: r.i.mackie@dundee.ac.uk (R.I. Mackie).

0045-7949/$ - see front matter © 2008 Civil-Comp Ltd and Elsevier Ltd. All rights reserved.
doi:10.1016/j.compstruc.2008.04.016

for this purpose. Using the UML, the essential features of an O-O finite element system can be described in a succinct and language-independent format. This makes it easier to spot recurring solutions in design.

The recurrence of a design solution does not itself prove that the solution is a good one. Moreover, there could be many repeating solutions for the same design problem. To establish best practices, the similarities discovered in the first phase were evaluated based on the criteria of flexibility and maintainability. These two criteria are particularly important because they are often touted as benefits of O-O programming. Sometimes, flexibility and maintainability represent opposing forces. A system designed for maximum flexibility may be more complex and therefore less maintainable. The goal is to balance these forces for better software quality on the whole.

In the final phase, the adopted design patterns were implemented in a finite element program first developed by the second author [7]. In this way, the patterns – which represent a synthesis of previous disparate research – could be tested as to their compatibility. This practical work also helped in understanding the forces shaping each pattern and the pattern's benefits and drawbacks. The insight gained during implementation proved useful in the documenting of the design patterns.

The documentation format follows broadly that used in [1]. The section heading names the pattern under consideration. The name of a pattern hints at its structure and is a useful communication aid. The purpose of the pattern is summarized under *Intent*. *Motivation* describes the context in which the pattern may be applied and the problems addressed. It also reviews alternative solutions in the literature.

The next section explains the proposed *Solution* with its rationale and underlying principles. The associations between participating classes are described under *Structure*. If necessary, object interactions are described under *Collaborations*. The UML is used in these two sections to illustrate the design. The benefits and drawbacks of applying the pattern are enumerated under *Consequences*. *Implementation* highlights implementation issues such as working with a particular language and potential pitfalls. Selected examples of the pattern in the literature are referred to under *Known Uses*. Finally, *Related Patterns* points the user to other patterns that may be part of the pattern under consideration.

3. Catalogue of design patterns

3.1. Model-Analysis separation

3.1.1. Intent

Decompose a finite element program into model and analysis subsystems.

3.1.2. Motivation

Earlier efforts in O-O finite element programming focused on defining model classes such as elements, nodes, boundary conditions, and materials. Little attention was paid to analysis-related tasks. Indeed, there was often no clear distinction between analysis-related code and model classes [8–10]. Without an appropriate scheme of organization, a finite element system can quickly become too large and complicated to maintain efficiently.

3.1.3. Solution

There are essentially two stages in finite element analysis. The first stage involves modelling the problem domain. The second stage involves analyzing the finite element model. It is natural therefore to decompose a finite element program into two major subsystems, one for modelling and the other for analysis. Model classes represent finite element entities such as elements, nodes, and degrees-of-freedom. The analysis subsystem is responsible for forming and solving the system of equations. The two subsystems should be loosely coupled. This means minimizing dependencies across subsystem boundaries.

Logically, analysis objects operate on model objects. Making the analysis subsystem dependent on the model subsystem is therefore a reasonable representation. This is also a more maintainable and flexible design. A stable subsystem should not be made dependent on a subsystem that needs to be flexible because that would make the latter rigid. The analysis subsystem should be amenable to changes and extensions. Model classes on the other hand are relatively stable. The analysis subsystem should therefore be dependent on the model subsystem.

3.1.4. Structure

Fig. 1 shows the three packages participating in this pattern and their dependencies. The Fe package contains model classes, while the `CalcCon` and `Solvers` packages together form the analysis subsystem. `CalcCon` classes represent different types of analysis. The `Solvers` package consists of mathematical classes for solving system equations. There is no coupling between `Solvers` and `Fe`.

3.1.5. Consequences

The following benefits may be obtained from applying this pattern:

(a) Decomposing an O-O finite element system into model and analysis subsystems helps clarify the system design. The clear division of responsibilities makes both maintenance and subsequent extensions of the system easier.
(b) Minimizing dependencies across subsystem boundaries reduces coupling and helps restrict the propagation of changes from one subsystem to another.
(c) Making the analysis subsystem dependent on the model subsystem allows the former to be changed and added to with little impact on the latter.

3.1.6. Implementation

The following are issues that should be considered in implementing the pattern:

(a) Some O-O languages – including C++, C#, and Java – facilitate the grouping of classes into logical namespaces. The classes in a namespace are typically packaged into the same physical assembly. Model and analysis classes should be

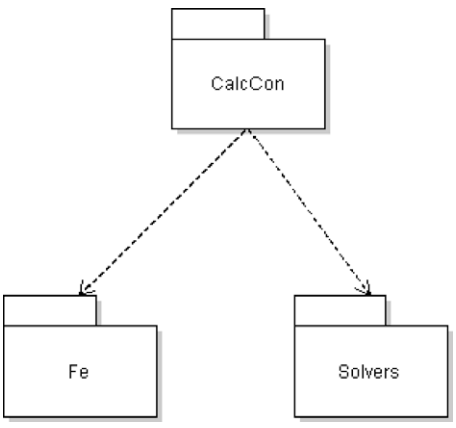

Fig. 1. Packages in the Model-Analysis separation pattern.

packaged into separate assemblies. This allows the implementation of a feature in one assembly to be changed without having to recompile the other.

(b) Dependencies across subsystem boundaries can be minimized using a *Façade* or a *Mediator* [1]. However, it is sometimes clearer and more efficient to have a direct link between a model object and an analysis object. Where this is the case, the client should not be dependent on a concrete class, but on an interface or an abstract class. This allows transparent switching of concrete implementations as far as the client is concerned.

3.1.7. Known uses

Rucki and Miller [11,12] were among the first to explicitly separate analysis classes from the model subsystem. The modelling and algorithmic frameworks interacted via partition (representing a group of elements) and degree-of-freedom objects.

Dubois-Pèlerin and Pegon [13] believed that a clear distinction between analysis-related classes and those related to the model is vital to implementing a flexible program. They assigned the responsibility for solving systems of equations to a `Problem` object. A `Domain` object represented an aggregation of model objects and was responsible for assembling the system of equations. Thus the `Domain` object acted as a kind of Façade or Mediator to decouple the `Problem` object from the model objects.

Other instances of this pattern can be found in [7,14–16].

3.1.8. Related patterns

A Façade [1] provides a single point of access to a subsystem, thus minimizing dependencies between subsystems.

A Mediator [1] provides centralized control of a group of classes. It defines functionality additional to those in the classes.

The *Modular Analyzer* pattern (see below) shows how to further decompose the analysis subsystem.

3.2. Model-UI separation

3.2.1. Intent

Separate methods and data related to the user interface (UI) from model classes.

3.2.2. Motivation

Most modern finite element systems have integrated graphical user interfaces. In O-O finite element programming, a graphical user interface could be implemented by adding UI-related responsibilities to the model classes [17,18]. But such an approach bloats model classes with incoherent responsibilities and increases the rigidity of the program.

3.2.3. Solution

UI-related responsibilities should be assigned to UI objects. UI classes should be grouped together in a subsystem that is dependent on the model subsystem. This allows the more volatile UI subsystem to be changed without affecting model classes. Naturally, there should be no coupling between the analysis and UI subsystems.

As in the Model-Analysis separation pattern, a Façade or Mediator may be placed between these two subsystems. But since there is a close correspondence between a model object and its UI representation, it would be clearer to implement a direct reference to the model object in the UI object.

3.2.4. Structure

Fig. 2 shows the dependencies between the `UI`, `Mesh` and `Fe` packages. `UI` contains classes such as `SubStruct`, `KeyLine`, and `KeyPt`. A `SubStruct` represents a (sub) domain of the finite element model. The `KeyLines` of a `SubStruct` describe its boundary. Each `KeyLine` is in turn defined by its `KeyPts`. These UI classes are used to build and manipulate a model on screen. Classes in the `Mesh` package are responsible for generating the mesh based on the on-screen model, creating element and node objects in the process. The class diagram in Fig. 3 shows the associations between the main model classes and their UI representations.

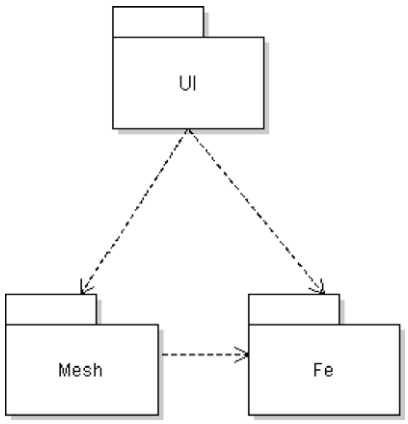

Fig. 2. Packages in the Model-UI separation pattern.

3.2.5. Consequences

This pattern has the following consequences:

(a) There is a clearer division of responsibilities. Model classes can remain coherent.

(b) Changes to UI classes do not propagate to the model subsystem. There is therefore flexibility with respect to the UI.

3.2.6. Implementation

In addition to the implementation issues in the Model-Analysis separation pattern, the developer should consider how to synchronize model and UI objects. The *Observer* pattern [1] may be used to relate a model object to its UI counterpart. The UI object can then be notified and updated automatically when there is a change in the model object, without having to make the latter dependent on the former.

3.2.7. Known uses

This pattern represents a variant of the *Model–View–Controller* pattern [19] well-known among O-O developers.

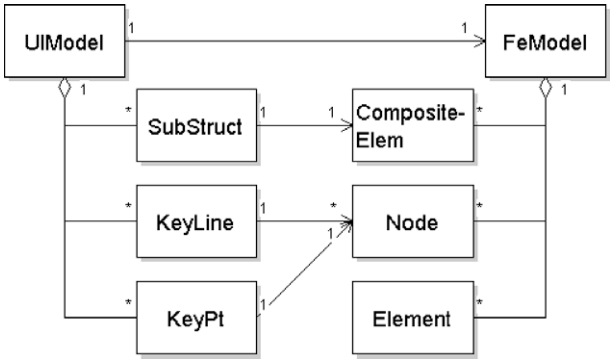

Fig. 3. Model classes and their UI representations.

In O-O finite element programming, the principle of separating graphical classes from model classes was proposed by Mackie [7] and Ju and Hosain [20]. The solution described above is a modification of Mackie's design.

A more general application of this pattern can be found in [21]. Input, output, and inter-process communication responsibilities were isolated in a module distinct from the modelling subsystem. This allowed flexibility in the choice of pre- and post-processors.

3.2.8. Related patterns

The Observer pattern [1] helps keep model and UI objects synchronized.

3.3. Modular element

3.3.1. Intent

Use object composition to define elements.

3.3.2. Motivation

The relative ease of adding new types of elements has been touted by many as one of the benefits of O-O finite element programming. Element classes can however be structured in different ways. The element class structure has an impact on the ease of adding new element types.

Earlier implementations used just one level of inheritance: all element types inherited from a single base class. An example can be found in [8]. This scheme allows some degree of flexibility but suffers from poor code reuse and tedious implementation. For example, 2D element classes have similarities that could be factored up into the base class to avoid code duplication. But 3D element classes would then have to override 2D-specific behaviour in the base class. And the problem of code duplication in the 3D element classes remains. This class structure also tends to result in an explosion in the number of element classes as the program is extended.

Mackie [7] and Pidaparti and Hudli [22] implemented a multi-level element hierarchy. Behaviour and attributes common to 2D elements were factored into an Element 2d class, which in turn derived from the base Element class. This scheme allows more code reuse. But there is a limitation: classes that do not share a common parent cannot share an implementation. Furthermore, deep hierarchies can be difficult to manage and debug. Extending a multi-level class hierarchy can lead to a sharp increase in the number of tiers and classes, exacerbating the problem of complexity and rigidity.

These limitations may have prompted Kong and Chen [23] to use multiple implementation inheritance in their element class structure. They defined each element class as a subclass of two classes, one representing problem type and the other representing geometric definition. However, multiple inheritance can complicate matters. The element class in this case is not a subtype of either of its parents; one cannot say that an element is a particular analysis type or a particular geometry. This conceptual anomaly could lead to maintenance problems later. Besides, some O-O languages – notably C# and Java – do not support multiple inheritance of implementation.

3.3.3. Solution

Consider object composition as an alternative to inheritance for functionality reuse. The approaches described above used the mechanism of implementation inheritance to facilitate code reuse. This is often referred to as *white-box reuse*, so called because a parent's internals are often visible to its subclasses. For the same reason, inheritance often breaks encapsulation, and tightly couples the implementation of a subclass to that of its parent. On the other hand, composition facilitates *black-box reuse* and preserves encapsulation. An element class may be decomposed into component classes by identifying points of possible variation [24].

In the context of computational solid mechanics, an element class may be assigned the responsibilities of forming element matrices and computing stresses. Both tasks involve the use of the strain–displacement matrix B and the constitutive matrix D. Each of these matrices represents a point of variation among element types. For example, a two-dimensional plane element may have the same shape functions and the same number of nodes and degrees-of-freedom as an axisymmetric element. But they have different B matrices. Likewise, a plane-stress element is differentiated from a plane-strain element by its D-matrix. These points of variation should be encapsulated in component classes.

Numerical integration schemes represent another point of variation that is orthogonal to the two described above. One approach to implementing numerical integration is to hard-code the integration schemes in the element class itself. In this approach, the element class contains switch–case or if–else statements to select the appropriate set of integration points based on the integration scheme selected and the element type. This bloats the element class, and makes adding new integration schemes tedious. The solution is to split up each branch of the decision structure into component classes that implement a common interface. Each class represents one integration scheme. New schemes can be added by defining new classes.

3.3.4. Structure

Fig. 4 shows the relations between the `Element` class and its components. The mechanism of interface inheritance or subtyping is vital to this pattern. An interface specifies a contract that implementing classes must adhere to. Unlike implementation inheritance, classes that implement an interface do not inherit any implementation details. It therefore preserves encapsulation. Clients of that interface need not know which implementation they are using.

Classes that implement `IIntegrPtTool` encapsulate different integration schemes. They are also responsible for calculating nodal stresses given the stresses at the integration points. Assigning these responsibilities to the same object ensures consistency between the integration points and the extrapolation functions. Classes that implement `IShapeFns` are responsible for computing shape function values based on integration point coordinates.

`IStrainDisplMatrix` and `IConstitutiveMatrix` define the interfaces for strain–displacement and constitutive matrix classes, respectively. They are used in the computations represented by the following equations:

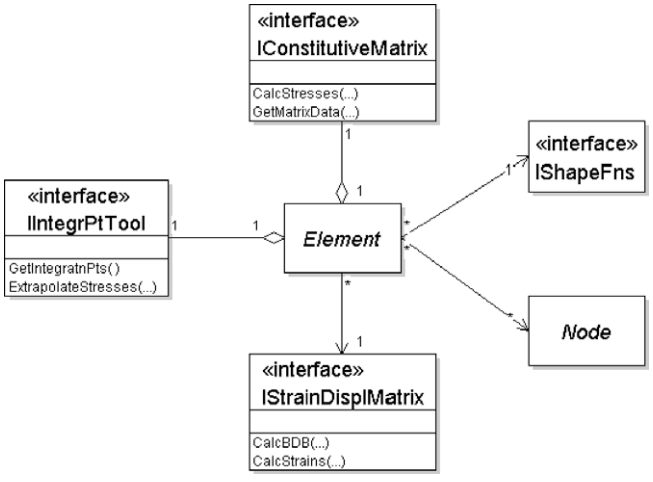

Fig. 4. Element component classes.

$$k^e = \int_{V^e} B^{\mathrm{T}} DB \, \mathrm{d}V^e, \tag{1}$$

$$\varepsilon = Bd^e, \tag{2}$$

$$\sigma = D\varepsilon, \tag{3}$$

where k^e is the element stiffness matrix, B is the strain–displacement matrix, D is the constitutive matrix, V^e is the volume of the finite element, d^e is the element's nodal displacement vector, ε is the strains at a point in the element, and σ is the stresses at that point.

Accordingly, B-matrix implementations are responsible for computing B^TDB and the strains, whereas D-matrix implementations convert strains into stresses.

3.3.5. Collaborations

The sequence diagram in Fig. 5 outlines the process of computing and assembling the element stiffness matrix into the global system of equations. Fig. 6 shows how `Element` uses the `IStrainDisplMatrix`, `IConstitutiveMatrix`, and `IIntegrPtTool` objects to compute and extrapolate stresses.

3.3.6. Consequences

Some consequences of applying this pattern are:

(a) Encapsulation is enhanced by forcing the element class to access functionality in its component classes through interfaces. Implementation changes in the element class will not propagate to component classes, and vice versa.
(b) Each component object has only a few responsibilities, meaning few axes of change. This keeps class hierarchies small and manageable.
(c) New element types can be defined by composing existing objects in new ways, which is generally easier than implementing new element subclasses.
(d) New component subclasses can implemented without affecting existing subclasses or the element class.

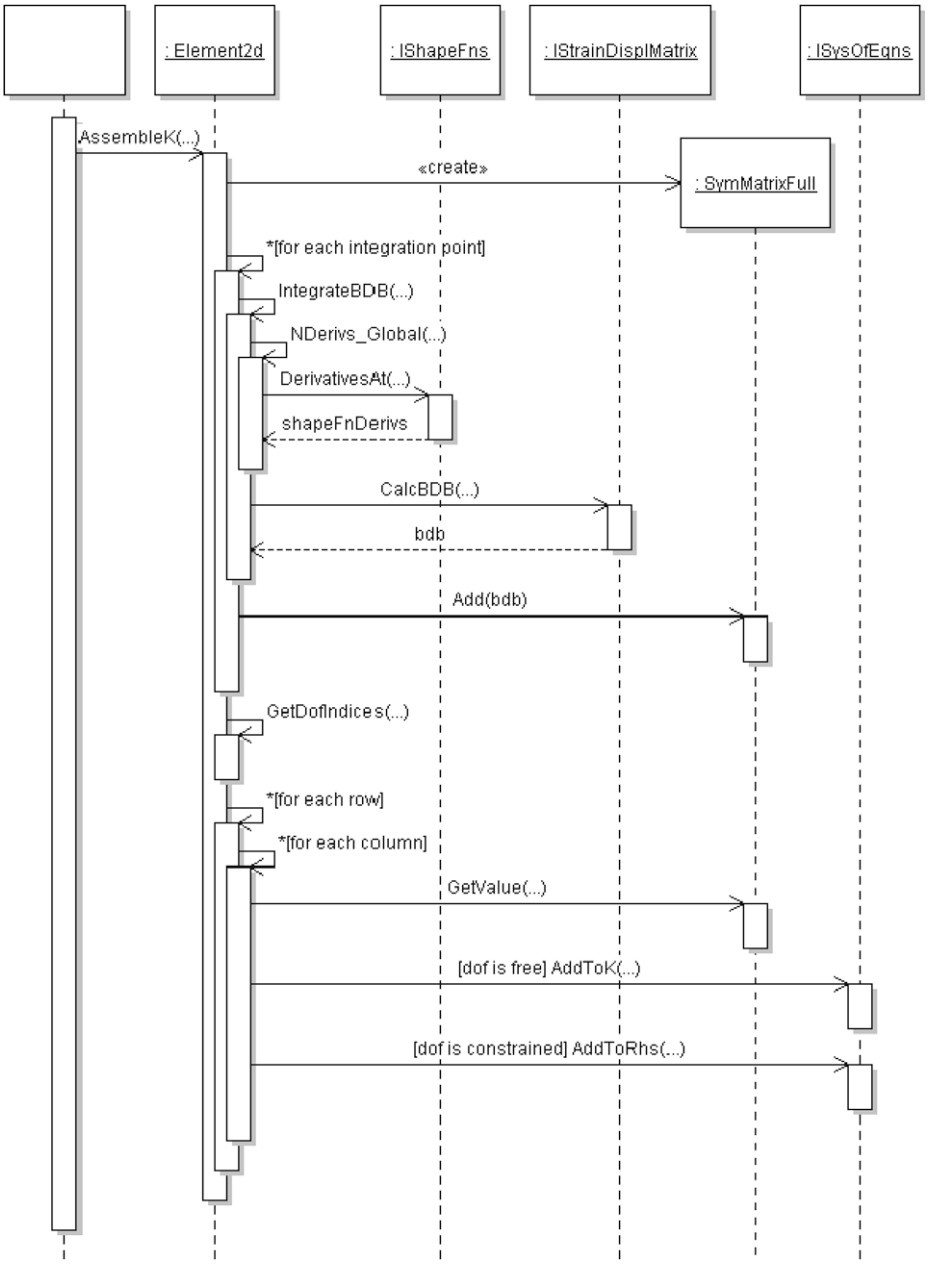

Fig. 5. Computing and assembling the element stiffness matrix.

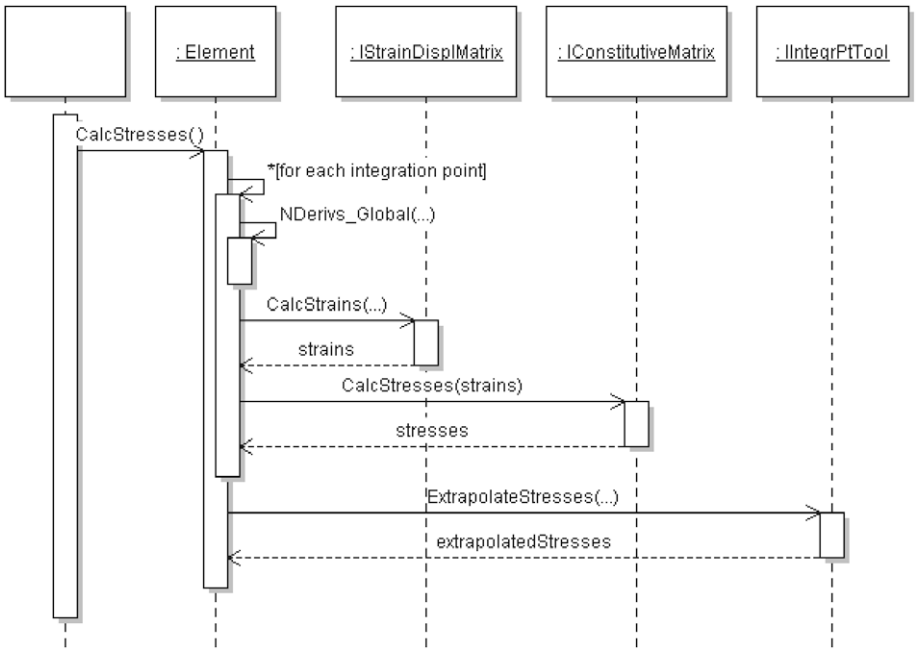

Fig. 6. Computing element stresses.

(e) Better code reuse is achieved since the implementation encapsulated in a component class is available to all element subclasses, whether or not they share a common parent. In fact, component classes can be used by non-element classes as well.
(f) The benefits of object composition are often counter-balanced by the difficulty of abstracting functionality from the target class into logical and coherent component classes. Comprehensibility may suffer as a result of haphazard abstractions.

3.3.7. Implementation

The following issues should be considered in implementing this pattern:

(a) Both C# and Java support interfaces. Pure abstract classes may be used in place of interfaces where there is no language support. In some cases, abstract classes with some default implementation may work just as well. For example, because differences among node subtypes are slight, an abstract parent class containing much of the implementation details may be used instead of an interface. This also makes implementing new subclasses less tedious. The judicious use of implementation inheritance can thus be useful.
(b) Component classes should not be implemented as mere data holders. Well-defined objects should have behaviour as well as attributes. Delegating responsibilities appropriately to component objects also keeps the element class from becoming bloated.
(c) `IStrainDisplMatrix` and `IShapeFns` may be implemented as *Singletons* [1] since they are stateless and can be shared among many element objects.
(d) For finite element analysis involving isotropic materials, `IConstitutiveMatrix` is best implemented as abstractions of constitutive equations instead of hard-coded matrix values. A constitutive matrix is calibrated with a material object that encapsulates material properties [12,14]. This scheme allows constitutive matrix classes to vary independently from material classes. Where anisotropic materials are involved, special constitutive matrix classes with hard-coded values could be implemented to avoid complex computations during initialization. The `IConstitutiveMatrix` interface facilitates both approaches without affecting the element class.

3.3.8. Known uses

Yu and Kumar [16] defined an element as a composition of an `AnalysisType` object and an `Interpolation` object. The former is essentially equivalent to the `IStrainDisplMatrix` and `IConstitutiveMatrix` combined, while the latter seems to combine the functionality of `IShapeFns` and `IIntegrPtTool`. Their design simplified the element class with fewer components at the expense of complicating the component classes. Kromer et al. [25] used a similar approach.

3.3.9. Related patterns

Some components can be implemented as Singletons [1]. The algorithms for computing element matrices and stresses may be implemented as *Template Methods* [1] in the base element class. Each component interface and its implementations are an application of the *Strategy* pattern [1].

3.4. Composite element

3.4.1. Intent

Implement a class to represent substructures such that clients can handle substructures and elements uniformly.

3.4.2. Motivation

A finite element model can be divided into several substructures to facilitate the use of domain-decomposition methods. A substructure could be represented simply as a container of element objects. This straightforward approach may suffice for some applications. But a substructure could be thought of as an element with many internal nodes [26] – a composite element. The implementation of a finite element program is simplified if substructures and simple elements could be treated uniformly. How can the concept of a composite element be modelled using O-O programming?

3.4.3. Solution

Define a common interface for substructures and simple elements, and implement a data structure in the substructure class to hold references to simple element objects. Clients that use the interface need not know if they are handling substructures or simple elements. A substructure usually forwards client requests to its elements. It may also be able to fulfil requests on its own, in which case the elements are not invoked.

3.4.4. Structure

`IElement` is the common interface for composite and simple elements (Fig. 7). A `CompositeElem` may contain any number of `Elements`; clients need not know.

3.4.5. Collaborations

As an example, Fig. 8 shows how `CalcConStatic`, which coordinates a static finite element analysis, uses the `IElement` interface to handle composite and simple elements uniformly.

3.4.6. Consequences

The benefits of applying this pattern are as follow:

(a) Clients of the `IElement` interface can be simplified since they handle substructures and simple elements uniformly.
(b) `CompositeElem` facilitates the grouping of elements into substructures for analysis using domain-decomposition methods. `CompositeElem` objects can be treated as independent entities for concurrent processing and distributed analysis.
(c) With `CompositeElem`, it becomes easier to manage groups of elements. For example, changing the material properties of a group of elements can be as simple as updating the `CompositeElem` to which they belong.

3.4.7. Implementation

Considerations in the implementation of this pattern include the following:

(a) It is possible but not necessary to put a reference in `Element` back to its parent. Doing so would, for example, allow an `Element` to notify its parent upon certain events. But maintaining two-way references can be complicated.
(b) In the design illustrated above, `Element` is of necessity an abstract class with concrete subclasses. This is to allow `CompositeElem` to contain different types of simple elements. If the containment link is between `CompositeElem` and `IElement` instead, the abstract `Element` class becomes unnecessary; concrete element classes can be placed on the same hierarchical level as `CompositeElem`.
(c) Letting `CompositeElem` contain `IElement` objects facilitates recursive composition, which may be useful in multilevel substructuring. On the other hand, `CompositeElem`'s implementation is simplified if it contains only simple elements. A type-specific `Add(...)` operation could be implemented in `CompositeElem` to restrict its components to `IElement` or simple element objects, as required.
(d) It is useful to let `CompositeElem` hold references to the nodes in its elements. `CompositeElem` can then invoke operations on the nodes without having to go through the elements. This is convenient and efficient during, for example, the numbering of the degrees-of-freedom in the subdomain represented by the `CompositeElem`. The nodes in a `CompositeElem` could be divided into interface (that is, shared with other `CompositeElems`) and non-interface nodes to facilitate domain-decomposition methods.

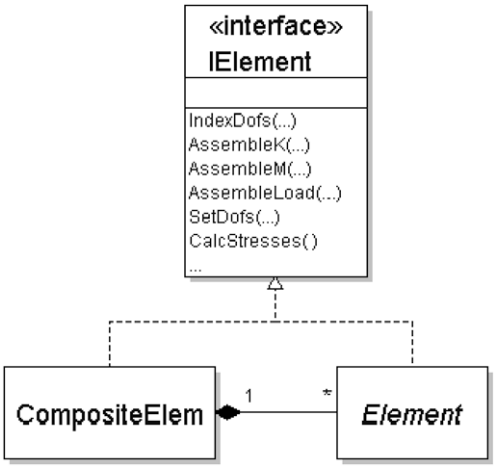

Fig. 7. `CompositeElem` and `Element` share the same interface.

3.4.8. Known uses

The class structure illustrated above is similar to the design used by Takahashi et al. [27]. An alternative that allows recursive composition can be found in [2,9].

Archer's [14] `SuperElement` is an element subclass that contained a `Model` object. This is a somewhat abstruse way of representing a substructure. In addition, bringing the `Model` object into the picture may increase rigidity.

3.4.9. Related patterns

Those familiar with O-O design patterns would recognize the Composite Element pattern as a variant of the *Composite* pattern [1].

The *Whole-Part* pattern [17] may also be used to represent a substructure. This pattern is easier to implement since the substructure and element classes do not share a common interface. But clients would not be able to handle substructures and elements uniformly.

3.5. Modular analyzer

3.5.1. Intent

Decompose the analysis subsystem into components.

3.5.2. Motivation

Early efforts in O-O finite element programming focused on finding appropriate objects to represent a finite element model. The design of analysis objects was largely neglected. As researchers sought to implement programs that could perform different types of analysis and with different solution algorithms, they began to pay greater attention to designing a flexible and extendible analysis subsystem.

Some implementations of the analysis subsystem relied on class inheritance to accommodate variations in analysis procedures and equation solution algorithms [11,16,22,28]. This produced monolithic objects with many responsibilities. It also tends to result in large class hierarchies. Maintenance and extensions are complicated as a result, and there is little scope for code reuse.

3.5.3. Solution

Decompose the analysis subsystem into component classes. As with the Modular Element pattern, object composition should be favoured over class inheritance.

There are at least two points of variation in the analysis of a finite element model. The first has to do with the type of analysis

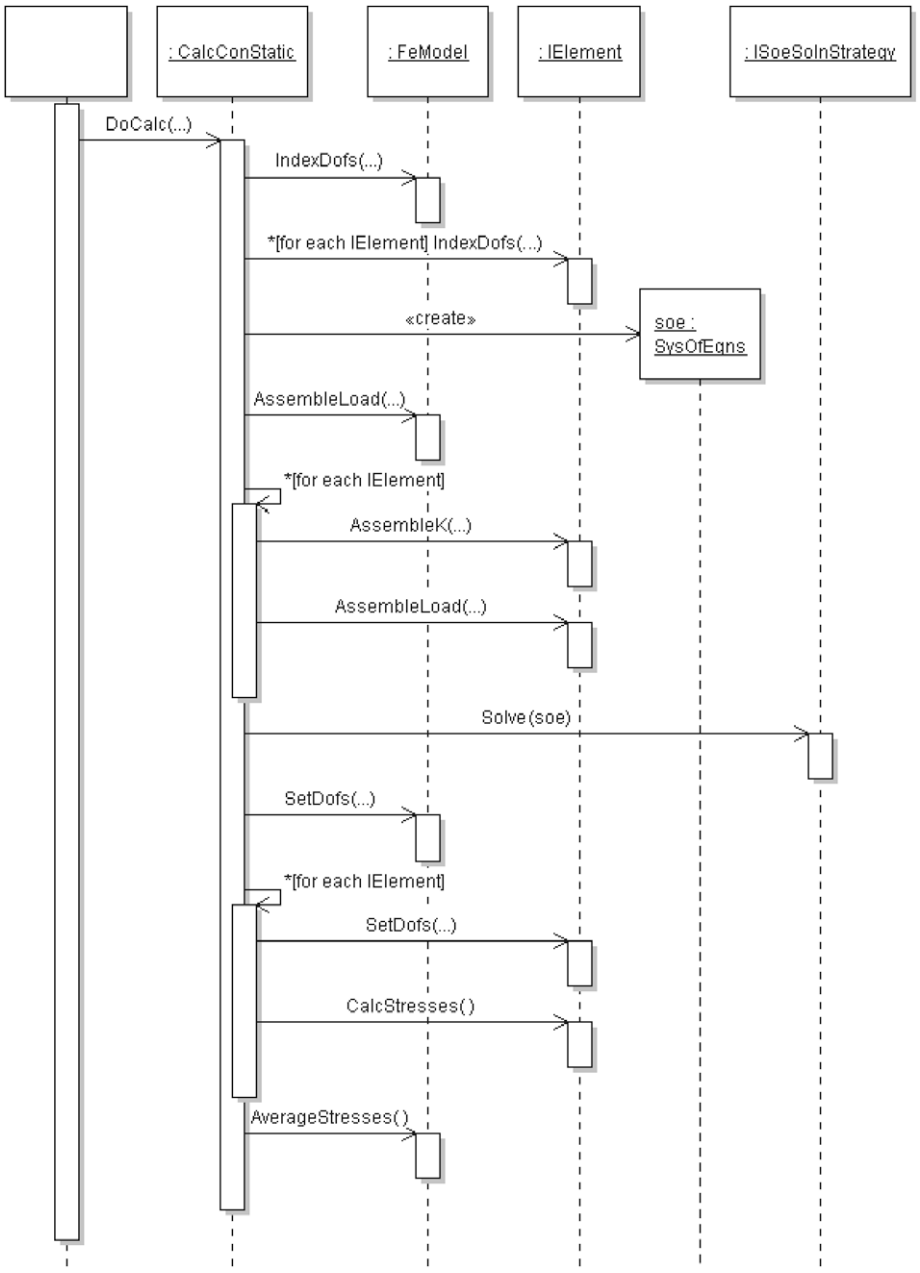

Fig. 8. `CalcConStatic` uses the `IElement` interface.

(static, dynamic, substructuring, etc.). This affects how the system of equations is formed and whether there are iterations. The other point of variation is in the solution algorithm applied to the system of equations. A proper decomposition of the analysis subsystem should allow these two aspects to vary independently. It makes sense to decompose along the lines of analysis type before finding objects to represent solution algorithms.

3.5.4. Structure

Classes that implement `ICalcCon` (Fig. 9) represent various types of analysis. For example, `CalcConStaticDD` analyzes a finite element model statically using one of several domain-decomposition methods, whereas `CalcConEigen` determines the model's lowest natural frequencies and vibration modes. Extending the program to perform, say, transient response analysis can be achieved by implementing a new subtype.

`CalcCon` objects may be associated with specific solution strategies. Different solution algorithms may be used with the same assembly procedure. The strategies encapsulate the second point of variation mentioned above. Concrete strategy objects are responsible for creating and using appropriate mathematical objects to solve the system of equations. For example, direct solution strategies make use of a `UtDUSolver` object to decompose and solve the system of equations, whereas conjugate gradient strategies make use of a `CGSolver`. Once the equations are solved, control returns to the `CalcCon` object, which then sets the degrees-of-freedom of the model.

3.5.5. Consequences

This pattern has the following benefits:

(a) Decomposing the analysis subsystem into components facilitates code reuse without complicating the main hierarchy.

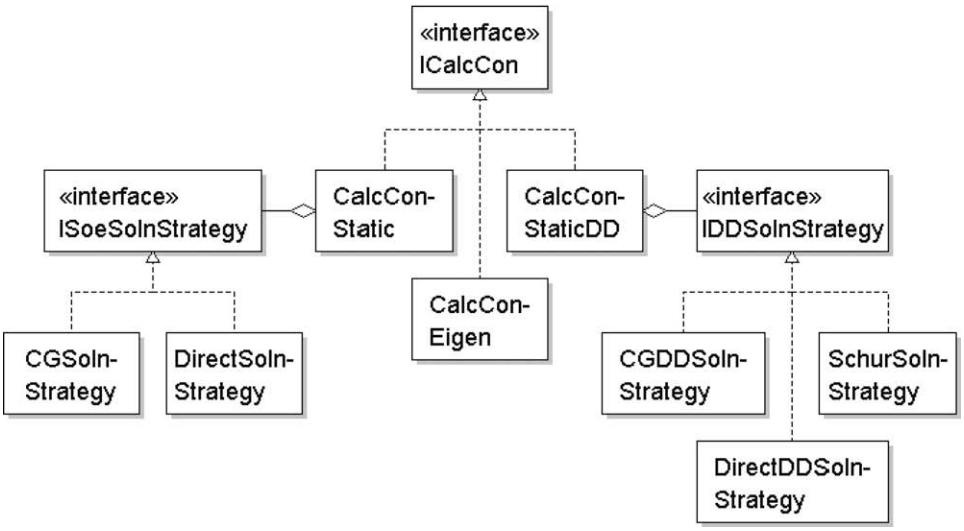

Fig. 9. `CalcCon` classes.

The main procedure encapsulated in a `CalcCon` class can be reused with different solution strategies. Mathematical classes like `CGSolver` and `UtDUSolver` can be used with `CalcConStaticDD` as well as `CalcConStatic`.

(b) The use of object composition and interfaces also increases flexibility. Existing classes are not affected by the addition of new solution strategies. Similarly, extending the system to perform, say, dynamic transient analysis has no impact on clients of the `ICalcCon` interface.

(c) There is greater coherence since component objects have only a small set of responsibilities. This eases maintenance.

(d) Since mathematical classes are independent of model objects, they may be replaced by general library classes implemented by specialists with minimal impact on the rest of the system.

3.5.6. Implementation

The following are some of the issues to be considered during implementation:

(a) `CalcCon` objects may be allowed to interact with model objects directly or forced to go through a Façade, such as `FeModel`. The former option is simpler but increases coupling between `CalcCon` and model classes.

(b) Broadly speaking, the analysis of a finite element model involves three stages. The first involves forming the system of equations. The system of equations is then solved. Finally the finite element model is updated with the results and stresses may then be computed. The first and last stages require interaction with the finite element model, whereas the second is mathematical. The analysis subsystem may on this basis be partitioned into two packages. One is dependent on the model classes; the other is not. Referring to the fore-going discussion, `CalcCon` classes and their associated strategies are placed in a package separate from classes such as `UtDUSolver` and `CGSolver`. This partitioning clarifies the design and eases maintenance.

(c) The system of equations should be encapsulated in a class separate from the solution algorithm. Classes representing systems of equations should implement a common interface. This would allow a solution algorithm to work with all storage representations. By the same token, a system of equations can be solved using any of the available algorithms. On the other hand, putting solution algorithm and storage scheme in the same class affords more room for optimization.

3.5.7. Known uses

Archer [14] decomposed their analysis subsystem along different lines. He factored out classes related to matrix handling, constraint handling, node reordering, and the mapping of degrees-of-freedom to equation numbers. These classes are used in assembling the system of equations.

McKenna [15] added components representing analysis types, integration schemes, and equation solution algorithms. Marczak [29] and Kopyssov et al. [30] also used this design.

Dubois-Pèlerin and Pegon [13] and Mackie [7] structured their analysis subsystems around two major points of variation: analysis type and solution algorithm. The design described under Structure is based on Mackie's.

3.5.8. Related patterns

This pattern applies the Strategy pattern [1] recursively to allow variations by analysis type and by equation solution algorithm.

4. Conclusion

The documentation of patterns in O-O finite element programming is a means of capturing and reusing expertise in this field. It brings together previous research into a common knowledge base. Researchers in this field can thus build on previous research instead of reinventing the wheel. It also represents a step towards unifying the different approaches to program design.

Five basic and widely applicable design patterns in finite element programming have been presented in this paper. They show how objects can be defined and organized for greater maintainability and flexibility. However, the details of each pattern are not meant to be prescriptive. Design patterns represent abstractions of design rather than source code. The variations in the literature show that the principles underlying each pattern can be applied in different ways. For example, the analysis subsystem can be decomposed in many ways. A finite element programming framework may have a more refined Modular Analyzer than would a specialized program. In general, finer decompositions offer greater

flexibility at the expense of being more difficult to understand. The developer has to decide on the most appropriate implementation, considering his particular requirements and constraints.

While there are many aspects of a complete finite element package that are not dealt with in the above design patterns, they do cover some of the most fundamental parts of a finite element package. Model-Analysis and Model-UI separate out the user interface, the finite element model itself, and the solution process. Modular element is relevant to the design of elements themselves and Composite Element deals with substructuring. Modular Analyzer is concerned with the solution methods themselves.

The patterns in this paper can certainly be improved upon through dialogue and the exchange of ideas. As stated, they by no means cover the whole field of O-O finite element programming. For instance, there may be patterns that are specific to certain applications of the finite element method. The requirements of parallel and distributed finite element analysis may call for a totally different set of patterns. Design patterns by definition represent collective knowledge and experience. The authors would therefore welcome all contributions to this knowledge base.

Acknowledgements

The first author gratefully acknowledges the financial support of the Nicholl–Lindsay Scholarship (University of Dundee) and the Overseas Research Student Award Scheme (Scottish Funding Council).

References

[1] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Reading, Massachusetts: Addison-Wesley; 1995.
[2] Liu W, Tong M, Wu X, Lee GC. Object-oriented modeling of structural analysis and design with application to damping device configuration. J Comput Civil Eng 2003;17(2):113–22.
[3] Fenves GL, McKenna F, Scott MH, Takahashi Y. An object-oriented software environment for collaborative network simulation. In: Proceedings of the 13th world conference on earthquake engineering, Vancouver, Canada; 2004.
[4] Eyheramendy D. High abstraction level frameworks for the next decade in computational mechanics. In: Topping BHV, Montero G, Montenegro R, editors. Innovation in engineering computational technology. Stirling UK: Saxe-Coburg Publications; 2006. p. 41–62.
[5] Mackie RI. Object oriented implementation of distributed finite element analysis in NET. Adv Eng Software 2007;38:726–37.
[6] Fowler M. UML distilled: a brief guide to the standard object modeling language. Boston: Addison-Wesley; 2004.
[7] Mackie RI. Object-oriented methods and finite element analysis. Stirling, UK: Saxe-Coburg Publications; 2001.
[8] Forde BWR, Foschi RO, Stiemer SF. Object oriented finite element analysis. Comput Struct 1990;34(3):355–74.
[9] Miller GR. An object-oriented approach to structural analysis and design. Comput Struct 1991;40(1):75–82.
[10] Baugh Jr JW, Rehak DR. Data abstraction in engineering software development. J Comput Civil Eng 1992;6(3):282–301.
[11] Rucki MD, Miller GR. An algorithmic framework for flexible finite element-based structural modeling. Comput Methods Appl Mech Eng 1996;136:363–84.
[12] Rucki MD, Miller GR. An adaptable finite element modelling kernel. Comput Struct 1998;69:399–409.
[13] Dubois-Pèlerin Y, Pegon P. Improving modularity in object-oriented finite element programming. Commun Numer Methods Eng 1997;13:193–8.
[14] Archer GC. Object-oriented finite element analysis. PhD Thesis, Civil Engineering, University of California; 1996.
[15] McKenna F. Object-oriented finite element analysis: frameworks for analysis, algorithms and parallel computing. PhD thesis, Civil Engineering, University of California; 1997.
[16] Yu L, Kumar AV. An object-oriented modular framework for implementing the finite element method. Comput Struct 2001;79:919–28.
[17] Zimmermann T, Bomme P, Eyheramendy D, Vernier L, Commend S. Object-oriented finite element techniques: towards a general purpose environment. In: Topping BHV, editor. Developments in computational techniques for structural engineering. Edinburgh, UK: Civil-Comp Press; 1995.
[18] Bettig BP, Han RPS. An object-oriented framework for interactive numerical analysis in a graphical user interface environment. Int J Numer Methods Eng 1996;39(17):2945–72.
[19] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. Pattern-oriented software architecture: a system of patterns. Chichester, UK: John Wiley & Sons, Ltd.; 1996.
[20] Ju J, Hosain MU. Finite-element graphic objects in C++. J Comput Civil Eng 1996;10(3):258–60.
[21] Balopoulos V, Abel JF. Use of shallow class hierarchies to facilitate object-oriented nonlinear structural simulations. Finite Elements Anal Des 2002;38(11):1047–74.
[22] Pidaparti RMV, Hudli AV. Dynamic analysis of structures using object-oriented techniques. Comput Struct 1993;49(1):149–56.
[23] Kong XA, Chen DP. An object-oriented design of FEM programs. Comput Struct 1995;57(1):157–66.
[24] Larman C. Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process. Upper Saddle River. New Jersey: Prentice Hall; 2002.
[25] Kromer V, Dufossé F, Gueury M. An object-oriented design of a finite element code: application to multibody systems analysis. Adv Eng Software 2004;35:273–87.
[26] Cook RD, Malkus DS, Plesha ME, Witt RJ. Concepts and applications of finite element analysis. New York: John Wiley & Sons; 2002.
[27] Takahashi Y, Igarashi A, Hirokazu I. Application of object-oriented approach to earthquake engineering. J Civil Eng Inform Process Syst 1997;6:271–8.
[28] Dubois-Pèlerin Y, Zimmermann T, Bomme P. Object-oriented finite element programming: II. A prototype program in Smalltalk. Comput Methods Appl Mech Eng 1992;98:361–97.
[29] Marczak RJ. An object-oriented programming framework for boundary integral equation methods. Comput Struct 2004;82:1237–57.
[30] Kopyssov SP, Krasnopyorov IV, Novikov AK, Rytchkov VN. Parallel distributed object-oriented framework for domain decomposition. In: Kornhuber R, Hoppe R, Pironneau O, Widlund O, Xu J, editors. Domain decomposition methods in science and engineering. Lecture notes in computational science and engineering, vol. 40. New York: Springer-Verlag; 2005.