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

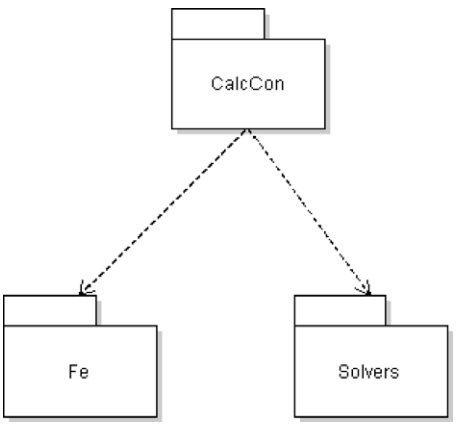

Fig. 1. Packages in the Model-Analysis separation pattern.