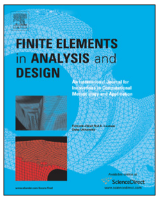
Contents lists available at ScienceDirect

Finite Elements in Analysis and Design

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



Nonlinear multiphysics finite element code architecture in object oriented Fortran environment

Zifeng Yuan, Jacob Fish*

Columbia University, New York, NY, United States

ARTICLE INFO

Article history:
Received 28 June 2014
Received in revised form
4 December 2014
Accepted 26 January 2015
Available online 21 February 2015

Keywords:
Finite element
Object-oriented programming
Fortran environment

ABSTRACT

The objective of the present manuscript is to describe a new architecture of the nonlinear multiphysics finite element code in object oriented Fortran environment hereafter referred to as FOOF. The salient features of FOOF are reusability, extensibility, and performance. Computational efficiency stems from the intrinsic optimization of numerical computing intrinsic to Fortran, while reusability and extensibility is inherited from the support of object-oriented programming style in Fortran 2003 and its later versions. The shortcomings of the object oriented style in Fortran 2003 (in comparison to C++) are alleviated by introducing the class hierarchy and by utilizing a multilevel programming style.

1. Introduction

The finite element analysis (FEA) software architecture is being constantly customized, upgraded and extended due to rapid developments of FEA capabilities, including new elements, local and nonlocal constitutive laws, contact and cohesive elements, and new multiphysics and multiscale capabilities. Adding new features or modifying existing codes is notoriously complex and error-prone. One of the key challenges for FEA code developers is to reduce the cost of developing and implementing new features. Thus, code reusability and extensibility are among the most important attributes of the FEA code structure. Reusability measures the ability that the FEA code to be used for other purposes with minor to no modifications. Extensibility means that the software can be easily extended and that the modification will have little to no effect on existing functionalities. Moreover, the FEA code developing process should be able to accommodate a teamwork.

The FEA codes, like programs in other areas, consist of data structures and algorithms. The data structure is a particular way of storing and organizing data blocks, while the algorithm is a step-by-step process which operates on data structures. Throughout the history of FEA software, data structures and algorithms have not significantly changed due to the fact that at the core, FEA has always been and still is the numerical solution of partial differential equations. However, the programming style that defines specific rules aimed at organizing data structures and algorithms has kept evolving over time.

For over 30 years since the inception of the finite element method, the FEA code structure was based on so-called procedure-oriented programming (POP) style mostly in the environment of FORTRAN 77. In the POP style, the FEA code is organized as a collection of relatively small procedures known as subroutines or functions. Each procedure consists of several commands that describe a particular algorithm. These procedures may have internal dependencies, i.e. procedures may be called by the other procedures. The complex data structures can be globally accessed throughout the FEA program. The POP programming style is schematically illustrated in Fig. 1.

FEA programs written in the POP style are usually tied to a specific algorithm. The lack of flexibility is magnified with increase in the number of subroutines. This lack of flexibility gives rise to the following shortcomings [1,2]:

(1) One has to understand the whole program structure before it can be modified;
(2) The dependencies between the subroutines are hidden and difficult to detect;
(3) It is hard to implement a new algorithm;
(4) The modification of subroutines may have unpredictable side effects;
(5) It is difficult for teamwork.

The philosophy behind the object-oriented programming (OOP) is based on the fact that the subroutines can be associated with data structures they operate on [1]. This self-contained entity

* Corresponding author.
E-mail address: fishj@columbia.edu (J. Fish).

http://dx.doi.org/10.1016/j.finel.2015.01.008
0168-874X/© 2015 Elsevier B.V. All rights reserved.

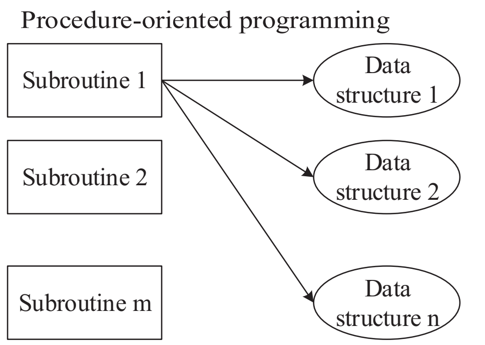


Fig. 1. Procedure-oriented programming: isolated subroutines with globally accessed data structures.

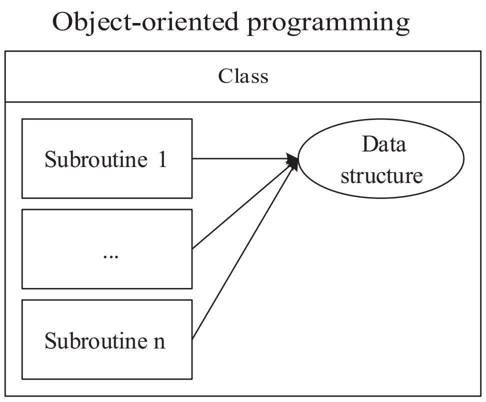


Fig. 2. Object-oriented programming: one object with its own data structure and subroutines.

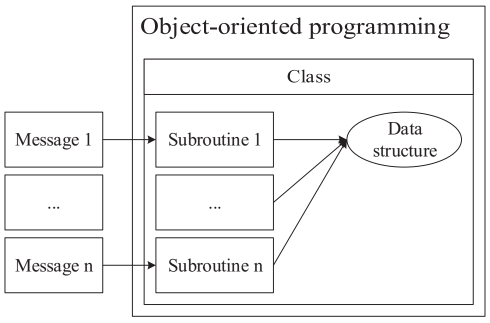


Fig. 3. The encapsulation of OOP: internal subroutines are called by sending messages.

Box 1–A general pseudo code for the derived type in Fortran.

```
MODULE mod_name
  TYPE:: type_name
      [define data structure]
    CONTAINS
        [define the interface of bounded subroutines]
    END TYPE type_name
CONTAINS
    [implementation of bounded subroutines]
END MODULE mod_name
```

Box 2–An example of the definition of the derived type in Fortran.

```
MODULE example_mod
    TYPE:: example_type
        INTEGER(KIND=4):: i
        REAL(KIND=8):: r
    CONTAINS
        PROCEDURE:: set_i=> set_example_type_i
        PROCEDURE:: set_r=> set_example_type_r
    END TYPE example_type
CONTAINS
    SUBROUTINE set_example_type_i(me, i_arg)
        CLASS(example_type):: me
        INTEGER(KIND=4),INTENT(IN):: i_arg
        me%i=i_arg
        RETURN
    END SUBROUTINE set_example_type_i
    SUBROUTINE set_example_type_r(me, r_arg)
        CLASS(example_type):: me
        REAL(KIND=8),INTENT(IN):: r_arg
        me%r=r_arg
        RETURN
    END SUBROUTINE set_example_type_r
END MODULE example_mod
```

Box 3–An example of sending a message to derived type in Fortran

```
PROGRAM example
  USE example_mod
  TYPE(example_type):: eg
  CALL eg%set_i(1)
  CALL eg%set_r(1.0D0)
  STOP
END PROGRAM
```

is called class, and objects are instances of a class [3]. For example, species of animals can be treated as a class, while a single animal of this species is an object of this class. Compared with the POP style, which requires developers to manage the interaction between the separate data structures and subroutines, the OOP encapsulates the specific kinds of data with the specific subroutines [4]. Accordingly, the basic concept of the OOP is abstraction that abstracts essential immutable qualities of the components as well as their methods into objects [2]. An object contains the data structure and bounded subroutines as illustrated in Fig. 2.

The OOP style requires that the data encapsulated in the object could not be accessed by external subroutines. The external subroutines can communicate with an object by sending messages only. In general, a message is a call of the bounded subroutines [5]. The mechanism of communication is schematically illustrated in Fig. 3.

The OOP style organizes different parts of the code as a set of objects with clear interfaces [6]. A well-defined object with its data structure and subroutines improves the understanding of the