Object-Oriented Programming in the Development of Containment Analysis Code

Tae Young Han[*a], Soon Joon Hong [a], Su Hyun Hwang [a], Byung Chul Lee [a], and Choong Sup Byun [b]
[a] *FNC Tech. Co. Ltd. SNU 135-308, San 56-1, Shinrim 9-Dong, Kwanak-Gu, Seoul, 151-742, S. Korea*
[b] *Korea Electric Power Research Institute, 65 Moonjiro, Yuseonggu, Daejeon, 305-380 , S.Korea*
[*] *Corresponding author: hanty@fnctech.com*

1. Introduction

After the mid 1980s, the new programming concept, Object-Oriented Programming (OOP) [1], was introduced and designed, which has the features such as the information hiding, encapsulation, modularity and inheritance. These offered much more convenient programming paradigm to code developers. The OOP concept was readily developed into the programming language as like C++ [1] in the 1990s and is being widely used in the modern software industry.

In this paper, we show that the OOP concept is successfully applicable to the development of safety analysis code for containment [2] and propose the more explicit and easy OOP design for developers.

2. OOP Design of Containment Analysis Code

In this section, the data structure of containment analysis code is described in detail and the process of the *Class* design and the features of OOP used in this code are presented in order.

2.1 Data and Function Structure

The general data and the function structures used in the containment analysis code are showed in Fig.1.

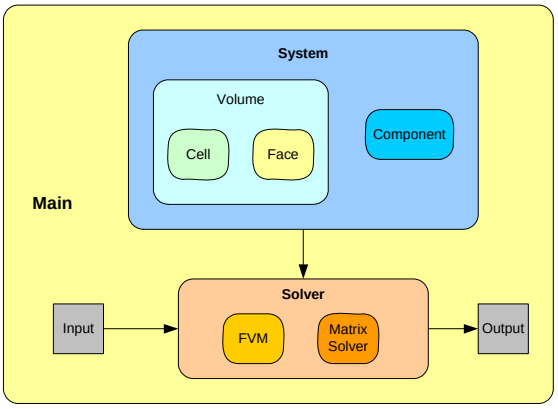

Fig. 1. Data & Function Structure for Containment Analysis Code

The data and the functions are roughly grouped into two classes, system and solver. Here, system is the object that could be accepted as containment or reactor system and has the information of volumes and components as like the geometrical data and the characteristic values of materials. On the other hand, solver is the set of the function for analyzing the system using numerical analysis methods, especially, a staggered semi-implicit finite volume method [2] in this research.

2.2 Class Design

In OOP, a matter of the highest priority is that the given system or data is analyzed, separated to some parts, and redefined by the name of *Class*. These processes are so-called *Class* design. After the basic *Class*es from the given data are written, a large scale *Class* can be designed using those. Then, the function and the logical relation between *Class*es using the written *Class* could be composed.

In the containment analysis code, the easiest way of the *Class* design is that the *Class*es are just written as the data structure of Fig.1 stands. For example, *Cell* and *Face Class* as the minimum data types in this numerical analysis method can be embodied. Here, *Cell* means scalar cell and *Face* means vector cell or momentum cell, and additionally it has the geometrical meaning as the boundary between neighboring cells. Then, *Volume Class* could be composed using these two *Class*es, which means a general control volume. In addition, *Component Class* is needed in order to simulate components as like valve. Lastly, *Volume* and *Component* belong together into *System Class* which is identical meaning to containment.

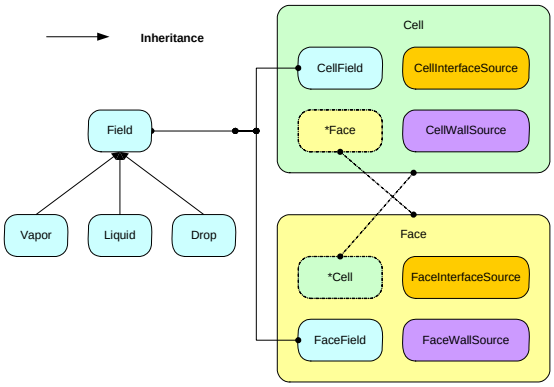

Fig. 2. *Class* Diagram of *Cell* and *Face*

Especially, Fig.2 shows the *Class* diagram among the *Cell*, *Face*, and *Field Class* composed in this code. *Cell* has memory addresses, namely pointer, of six *Face*s in the case of rectangular geometry and also owns the three objects of *Field Class*, vapor, liquid and drop, including material data. Likewise, *Face* has two pointers of neighboring *Cell*s and three independent objects of *Field Class*.

When these *Class*es are actually written, *Field*, *Face* and *Cell Class* as small scale *Class* are firstly designed,

and then *Volume* and *System Class* as larger scale *Class* using the previously written *Class*es are composed. This manner can make the more efficient development than the old manner that the development proceeds from the whole system to a part.

2.3 Data Hiding and Encapsulation

The features of OOP to be considered in *Class* design are data hiding and encapsulation. These are to make the interface which permits or forbids external functions to get access to other *Class*. Additionally, they mean that global or common variables are used as few as possible. Because the appropriate data encapsulation can increase the reliability of code and the efficiency of maintenance, it was widely accepted as modern programming method.

In this code, all *Class*es have the *public* member functions which external functions or objects have access to. For example, in order to update the member variables inside *Cell* objects, an external function just call a *public* member function, updateField(), defined in *Cell Class*. In other words, an external function or *Class* need not get direct access to the member variables of *Cell Class*.

From the following simple programs written by old manner and by the concept of encapsulation, one can find out the difference of two manners.

Old Type	`Cell cells[];`
	`updateCell() {`
	`  for( int i=0; i<N; i++ ) {`
	`  cells[i].xxx =`
	`  cell[i].aaa+cell[i].bbb;`
	`  }`
	`}`

OOP Type	`updateCell( Cell cells[] ) {`
	`  for( int i=0; i<N; i++ ) {`
	`  cells[i].updateField();`
	`  }`
	`}`
	`public Cell::updateField() {`
	`  xxx = aaa+bbb;`
	`}`

Fig. 3. Sample Programs (I) by old type and OOP type

In the old type, all the data are opened and a external function can get direct access to those. But, this type has the risk that programmer doesn't know where the values of variables were changed and the inconvenience that one should read all contents of the function. In OOP type, however, the external function has only to call the *public* function, updateField(), without direct access to the variables of *Cell Class*.

2.4 Modularity and Portability

Because the well designed *Class*es using the above stated OOP features has independence to other *Class*es, the member data inside *Class*es can be efficiently transferred to other *Class*es or functions and can be easily exported to other codes. Also, when the code needs to be modified, programmers can accomplish the purpose by slight efforts.

Fig. 4 shows two examples written by old manner and by the concept of OOP.

Old Type	`solveSystem( cell[], face[],` `  component[] , int N, float x );`
OOP Type	`solveSystem( system );`

Fig. 4. Sample Programs (II) by old type and OOP type

All data in the program of old type have to be individually transferred to external functions through the parameters or the form as global or common variables. But, the only parameter in the program of OOP type is a single System object including all data such as *Cell* and *Face* objects. Hence, if *System Class* is designed as the containment including all volumes and components and the other parts inside containment are implemented to their equivalent *Class* as well, the each *Class* have the independence and the high portability.

3. Conclusions

The safety analysis code for containment was successfully implemented using the concept of object-oriented programming. The given containment system is separated to small parts and redefined by some *Class*es such as *Volume*, *Component*, *Cell* and *Face* according to the principal of data hiding and encapsulation. Then, it was defined as *System Class*.

Consequently, the each *Class* has the modularity and portability and the entire code can be efficiently implemented. The code was verified by obtaining the reasonable calculation results.

Acknowledgment

This study was performed under the project, "Development of safety analysis codes for nuclear power plants" sponsored by the Ministry of Knowledge Economy

REFERENCES

[1] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley Pub. Co., 3rd Edition, 2000.
[2] S.J. Hong, S.H. Hwang, T.Y. Han, B.C. Lee, C.S. Byun, A Staggered Semi-implicit Finite Volume Method to Solve 3-dimensional Containment Phenomena Based on Cell and Face Porosity, Transactions of the Korean Nuclear Society Autumn Meeting, Pyeong Chang, Korea, 2008.