研究論文集 第33輯 2003年 12月 pp. 27~38

확장이 쉬운 구조의 객체지향 유한요소해석 프로그램

남용윤*

An Object-oriented Finite Element Analysis Program Architecture of Easy Extension

*Yongyun Nam**

ABSTRACT

The object-oriented programming languages are widely used in the modern software engineering. The procedural type FEM codes are still used because it is very hard and cost consuming job to re-code them into object-oriented programs. In this study, a FEM program was newly designed and coded with C++, an object-oriented language. Also a special programming technique, auto-loading technique was developed for open structured program, with which the extension and maintenance can be carried out easily. For example, the additions of element libraries to an existence FEM program do not require any modifications of the program.

* 구조연구부 구조안전그룹 책임연구원

1. 서 언

C++, JAVA 와 같은 객체지향 전산프로그램 개발 언어는 절차중심 언어와 비교하여 단점도 있지만 소프트웨어 공학 측면에서 많은 이점이 있다. 제일 큰 장점은 코드의 재사용이고, 두 번째는 프로그램 대상을 보이는 대로 자연스럽게 모델링할 수 있다는 점이다. 현대의 소프트웨어 패러다임은 "Search and Patching"으로 필요한 코드를 찾아 엮는 방식으로 진행된다. 객체지향 언어를 사용한 대형 복잡 프로그램 개발의 생산성이 매우 높고, 유지/보수가 쉬운 것은 객체지향 언어의 이 두 가지 이점에 기인한다.

한편 일반적인 목적의 전산프로그램 개발의 경우 객체지향 언어로 작성된 참고할 만한 코드가 많이 있지만 공학 분야에서는 그렇지 못하다. 객체지향 언어가 공학분야에서는 아직 활용이 활발하지 않기 때문이다. 본 연구에서 개발한 유한요소해석 전산프로그램도 마찬가지로 아직 활용할 코드가 많지 않다.

유한요소해석법을 전산프로그램으로 구현하는 것은 복잡한 문제인데, 80년대에 인공지능 기법으로 유한요소해석 전산프로그램을 개발하려는 연구가 수행된바 있다. Bruce[1]등은 처음으로 객체지향언어를 유한요소법 프로그램에 적용하였고, 1990년대에 들어 객체지향 언어를 이용하여 유한요소 해석 프로그램을 개발하는 연구가 다수 수행되었으며[2]-[7], 객체지향 언어를 유한요소해석 전산프로그램 개발에 사용하도록 권고되었다. 이 기간 동안의 연구동향은 참고문헌 [7]-[8]에 정리되어 있다. 여기서는 최근의 연구에 대해서 간단히 살펴본다.

Archer[7] 등은 수치계산과 해석 알고리즘과 관련한 해석객체와 모델 객체를 분리한 객체지향 유한요소해석 프로그램 구조를 연구하였다. 모델객체는 절점, 요소 등과 같은 다수의 유한요소 모델링에 관한 객체를 포함하는 메가객체이고, 수치해석 객체는 연립방정식을의 해를 구하기 위한 객체이다. 이들은 모델 개체와 해석객체간의 연결을 위한 별도의 클래스(Map 클래스)를 도입하였고, 제한조건, 행렬, 재배열 등을 취급하기 위한 핸들러 클래스를 만들었다. 이 구조의 장점은 해석알고리즘과 모델이 분리되어 있어 프로그램 수정시 서로 미치는 영향을 최소화하거나 관리가 잘 되도록 한 점이다. Lichao와 Kumar[8]은 Archer[7]의 경우와 비슷하게 모델 객체와 해석객체를 분리한 구조를 제시하였다. 별다른 특징은 없지만 요소 클래스에 수치적분 형태와 요소 기하형태(예: 보, 평면변형, 평면응력, 봉 등)를 각각 별도로 모델링한 클래스를 포함시켜 요소 개발의 효율성을 높였다. Modak과 Sotelino[9]는 기존 개발된 객체지향 유한요소 프로그램을 이용하여 동적해석을 병렬로 수행하는 프로그램으로 확장하는 방법을 제시하였다. NieKampr와 Stein[10]은 유한요소 프로그램은 아니지만 p-적응 메쉬 생성에 객체지향 방법을 적용하였다.

본 연구에서도 객체지향 유한요소해석 프로그램을 위한 하나의 구조를 제시한다. 앞의 연구자들과 마찬가지로 유한요소 모델과 해석객체를 분리한 구조를 채용한다. 또한 프로그램의 기능 추가가 기존의 코드를 수정하지 않고 자동으로 수행되는 기법을 적용하여 프로그램의 확장성을 높였고, 몇 가지 특수한 기능을 부여하였다.

2. 프로그램 개념설계

모듈화를 통하여 프로그램 개발 및 보수/유

지의 독립성을 높이고, 모듈간의 자료전달을 효율성과 편리성을 위하여 객체의 주소를 관리하는 클래스 사용한다(필요한 자료 주소 클래스를 상속하면 그 자료를 이용할 수 있도록 함). 그리고 프로그램의 확장성을 최우선으로 고려하여 기능의 추가가 쉽게 가능하도록 설계한다. 즉, 다양한 입력파일 형식을 지원할 수 있도록 자료 입력 기능 추가, 다양한 해석을 수행할 수 있도록 해석 종류 추가, 요소 추가 등을 기존의 코드 수정 없이 추가할 수 있도록 한다.

요소, 절점, 하중 등의 유한요소 모델의 대상이 되는 개체뿐만 아니고 해석 알고리즘도 객체로 모델링한다. 즉, 프로그램에 참여하는 모든 대상을 개체로 모델링한다.

그림 1과 같이 프로그램의 구성을 Interface, Model, solver, job, Utility, Results 등의 6개의 콤포넌트로 설계한다(Monitor, CrackModeler, Post는 추후 개발될 예정인 콤포넌트임). 그림 1에서 나타난 콤포넌트간의 관계는 상징적인 것으로 실제의 관계를 나타낸 것이 아니다. 예를 들어 solver 모듈은 job 모듈의 맴버로 이용되고, Utilities로 분류된 Matrix 클래스는 다른 객체에서 수시로 사용되는 클래스이다.

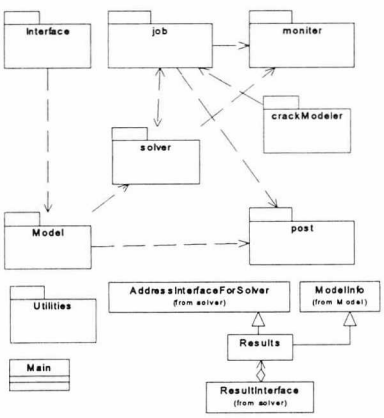

그림. 1 프로그램 모듈 구성

3. 프로그램 모듈 및 클래스

3.1 Utility 모듈

Utility 모듈은 실제의 모듈이 아니고 프로그램 관리상 설정한 모듈로 다른 모듈의 클래스에서 사용할 클래스를 묶어 관리하는 편이상의 모듈이다. Utility에는 다음 5개의 클래스가 있다.

FileManager : 파일의 생성과 파일명 작명을 위한 클래스.

Log : 에러 메시지나 객체의 어느 곳에서나 파일에 자료를 출력할 가능케 해주는 클래스로 출력은 Log 파일이나 정해진 출력 파일에 출력을 해주는 다양한 함수가 내장된 클래스.

Matrix : 다양한 행열 연산을 위한 클래스로 연산자가 이중으로 되어있어 행렬 연산을 간단하게 기호적으로 표현이 가능.

SymmetricMatrix : 대칭행렬 클래스로 Matrix 클래스와 비슷함

GaussIntegration : 수치적분을 위한 클래스. 이 클래스에는 기하학적인 형태에 따른 적분점 위치와 가중치가 정적변수로 저장되어 있다.

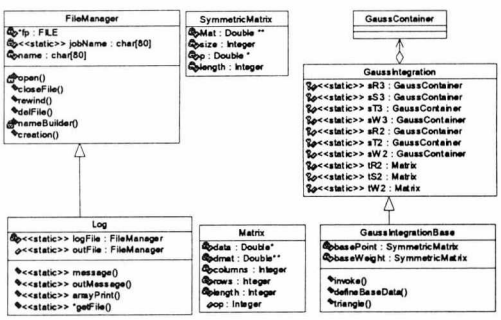

그림. 2 Utility 모듈

3.2 Interface 모듈

이 모듈의 중심이 되는 클래스는 추상 클래스인 ***Interface***로 ***FileManager, DataConvert*** 클래스의 객체를 멤버속성으로 포함하고 있다. ***FileManager*** 객체는 입력 파일인데 파일은 ***FileManager***의 멤버 함수에 의하여 생성되고 파일명은 ***FileManager***의 정적 맴버변수 *name*에 저장되어 이름을 이용하여 생성된다.

데이터를 직접 읽고 데이터를 파싱하는 것은 *DataConvert* 클래스가 담당한다. 입력자료는 <Tag>를 사용하여 표시되며, 모든 자료를 문자 형식으로 버퍼에 받아들인 후 파싱하여 필요한 형으로 변환하여 사용한다.

Interface로 클래스는 데이터를 입력작업을 수행하는 함수 *doReadingWork()*를 가상함수로서 갖고 있다. 실제로 데이터를 읽어드리는 클래스는 ***Interface*** 클래스를 상속하여 파일의 내용과 순서에 맞게 읽어드리는 함수 *doReadingWork()*만 작성하면 된다.

이와 같이 인터페이스 클래스는 다른 프로그램에서도 재사용이 가능한 공통적인 클래스이다. 따라서 개발자는 파일의 생성, 에러메시지 처리, 데이터 처리 등에 시간을 낭비하지 않고 프로그램을 개발할 수 있다. 현재는 본 프로그램의 기준 입력 인터페이스인 ***DefaultInterface***가 설치되어 있다. 그리고 ***AutoInterface*** 클래스는 다양한 인터페이스를 자동으로 추가하기 위한 클래스로 작동방법은 요소 자동 추가를 위한 클래스와 같은 기술을 사용한다(4장에서 설명됨).

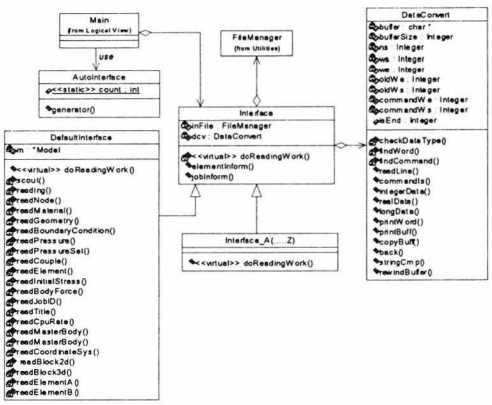

그림. 3 Interface 모듈

3.3 Job 모듈

유한요소해석에는 예를 들어 정적구조 해석, 동적 구조해석, 기타 다양한 해석 종류가 있다. ***Job*** 모듈은 이러한 다양한 해석방법을 설치를 용이하게 할 수 있도록 도입된 모듈이다. 개개의 해석 클래스는 *Jobs* 클래스를 상속하는데, *Jobs*는 멤버함수 *doJob()*이 가상함수이기 때문에 추상 클래스이다. 해석 클래스는 Job 클래스를 상속받아서 가상함수 *doJob()*와 *printJobInform()* 함수를 작성하면 된다. *AutoJob* 클래스는 자동으로 해석클래스를 설치하기 위한 것이다.

List 1 은 정적해석의 한 예를 보여주는 프로그램 코드이다. 이 코드를 기준 틀로 하여 List에서 진하게 표시된 부분만 새로 추가하는 해석방법에 따라 코드를 작성하면 기존의 프로그램 수정 없이 코드를 설치할 수 있다(이에 대해 자세한 것은 4장에서 설명됨). 좀 더 List를 자세히 보면 *doJob()* 함수는 실제로 해석수행을 지시하는 함수로 *FrontalSolvers* 객체를 생성하고 선택사항에 따른 작업을 수행

한 후 해를 구하는 것으로 되어 있다. 다른 복잡한 해석에서는 이 부분의 코드가 길어지고 또 필요한 함수를 추가하여 코드를 구조적으로 만들 수도 있다. *printJobInform()* 함수는 해석의 종류에 대한 정보를 출력하는 함수이다.

코드의 마지막 부분은 이 클래스를 자동으로 설치하기 위한 부분이다.

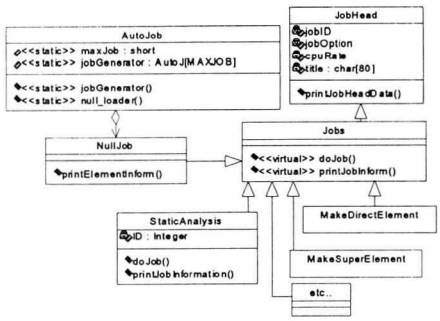

그림. 4 Job 모듈

```
#if !defined STATICANALYSIS
#define STATICANALYSIS
#include "autojobs.h"
#define IDNO_0 0
class StaticAnalysis:public Jobs
{
        private:
                int ID;
        public:

          StaticAnalysis(){ID=IDNO_0;}
                ~StaticAnalysis(){};
         void doJob();
         void printJobInform();
};
#endif

#include "StaticAnalysis.h"
#include "frontalsolvers.h"
#include "AddressInterfaceSolver.h"
void StaticAnalysis::doJob()
{
   FrontalSolvers *sol ;
   sol=new FrontalSolvers();

   if(jobIdOption>0)sol->protocol.send_SuperElement=sol->protocol.send_NeedSuperElSol='Y';
   sol->protocol.send_NeedSolRecord='Y';
   if(sol->protocol.send_SuperElement=='Y')AddressInterfaceForSolver::super->loadFromFile();
   sol->doNormalSolver(cpuRate,jobId );
   delete sol;
}
//----------이 코드를 자동으로 설치하기 위하여 추가되는 코드----------
void StaticAnalysis::printJobInform()
{
            Log::outMessage(ID," : Static analysis");
}
static Jobs* StaticAnalysis_loader(){
            return (Jobs*)new StaticAnalysis();
}
static AutoJobs staticAnalysys_o(StaticAnalysis_loader,IDNO_0);
```

List. 1 Static Analysis 클래스

3.4 Model 모듈

Model 모듈은 해석 대상 및 하중 기타 유한요소해석에 관련된 유형 무형의 모든 개체의 모델링을 포함하는 메가 클래스이다. 이 모듈에 참가하는 대부분의 클래스는 상속이 아닌 포함관계로 되어있다. 단 *ModelInfo* 클래스는 예외로 모델에 관한 정보가 들어 있는 곳이다.

요소에 관련한 *Element* 클래스는 요소 개발의 편이성과 효율성을 고려하여 구성되었다. *Element* 클래스는 요소강성 행렬 계산 시 필요한 외부자료를 얻기 위하여 *AddressInterfaceElement* 클래스, 수치적분에 필요한 자료를 얻기 위하여 ***Utility*** 모듈의 *GaussIntegration* 클래스를 상속받는다. 이 클래스는 하나의 가상함수를 갖고 있기 때문에 추상 클래스로 각 요소는 이 추상 클래스를 상속받아 추상 함수를 정의하면 된다. 따라서 요소의 개발을 쉽게 할 수 있다. 추상 클래스에는 요소 개발에 필요한 다양한 함수가 정의되어 있어 요소 개발 시 편리하게 사용할 수 있다.

AutoElement 클래스는 요소를 자동으로 설치하기 위한 것으로 4장에서 설명된다.

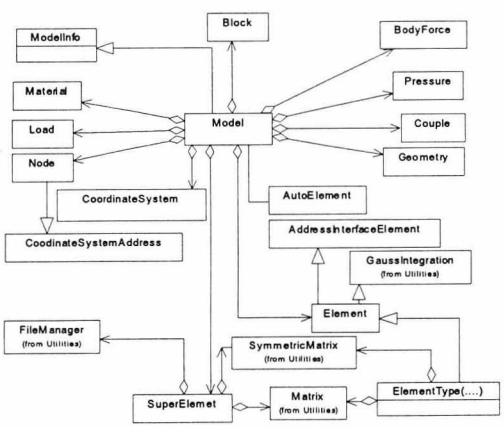

그림. 5 Model 모듈

3.5 Solver 모듈

Frontal 방법을 근간으로 하여, 기능에 따라 나누어진 클래스로 구성하여 보수/유지가 쉽도록 한다. 이 모듈의 중심은 *FrontCenter* 클레스로 전통적인 Frontal 기법에 관한 함수와, 시스템 행렬을 멤버 변수로 갖고 있다. 유한요소 해석모델은 주소관리 클래스 *AddressForSolver*를 통하여 이루어지고 연산결과는 결과 주소관리 클래스 *ResultInterface*를 통하여 외부로 전달된다. *EliminationTime* 클래스는 요소에서 소거할 절점의 시기를 결정하는 클래스이고, *OutCoreManager*는 Frontal 기법에서 소거된 식에 관한 자료를 파일에 기록할 때 이를 담당하는 클래스로 컴퓨터의 메모리 량을 고려한 버퍼를 생성하는 기능을 갖고 있다. *SolverProtocol*은 해석의 선택사항을 표현하는 클래스이다(이 선택에 따라 다양한 종류의 해석 기능이 선택됨). *FrontSpaceManager*는 시스템 행렬의 비어있는 위치를 관리하는 클래스이다.

다음은 연산 효율과 해석의 규모에 상관없이 이 코드를 적용할 수 있는 기능에 대해 정리한 것이다.

- 반복되는 요소의 강성행열 계산 생략 기능.
- 시스템 행렬의 크기를 해석대상의 규모에 따라 정확히 계산하여 메모리 낭비를 없애고, 프로그램 수정 없이 작은 문제부터 큰 문제까지 해석할 수 있게 함
- 컴퓨터의 물리적인 잔여 메모리 양을 조회하여 이를 참고한 대형 버퍼를 만들어 가능하면 외부 기억장치를 이용하지 않고 모든 연산을 수행하게 하고, 규모가 큰 문제는 일단 버퍼에 역대입을 위한 자료를 저장한 후 버퍼가 차면 외부 기억 장치에 자료를 기록함.

· 강성행렬의 요소가 0인 경우는 철저히 연산을 생략하여 계산 시간을 줄임(강성행렬의 이산도가 큰 경우 매우 효과적임).

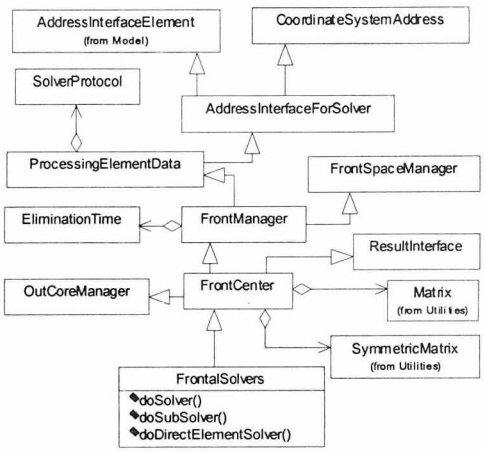

그림. 6 Solver 모듈

4. 프로그램 확장을 쉽게 하는 기법

입력자료 형식, 해석 종류, 요소 등을 기존의 프로그램 수정 없이 추가할 수 있는 기법을 개발하였다. 세 가지 모두 같은 기법을 사용하고 있기 때문에 요소 자동 추가를 위한 방법을 대표적으로 설명한다.

이 기법은 C++의 다음의 특성을 이용하여 개발된 것이다.

· 부모 클래스로 생성된 객체는 부모클래스로 생성된 참조자로 참조가 가능함(역은 성립하지 않음)

. 정적 객체 생성은 프로그램이 수행되기 전에 미리 수행됨

. 함수참조 포인터 변수를 사용할 수 있음

4.1 AutoElement 클래스

프로그램 List를 예로 하여 이 기법을 설명한다. List 2에 요소자동추가 클래스 코드를 나타냈다. 이 클래스에서 중요한 역할을 하는 멤버변수는 elementGenerator[ ]로 *Element* 형태의 객체에 대한 포인트를 반환하는 함수를 저장하는 배열이다. 앞서 기술한 이 배열의 형은 AutoEl 로 코드에서 "typedef Element* (* AutoEl)();"에 의하여 정의되어 있다. 또한 이 elementGenerator[ ]는 정적변수(자바에서는 클래스 변수라 함)로 *AutoElement* 클래스의 객체가 많이 생기더라도 한번만 생성되는 특성이 있다(*AutoElement*로부터 생성되는 모든 객체를 통하여 공용으로 사용될 수 있음).

AutoElement 에는 두 개의 생성자가 있는데 디폴트 생성자는 elementGenerator[ ]에 nullElement를 대입하기 위하여 사용된다. 나머지 생성자는 특정 요소를 설치하는 함수의 이름을 매개변수로 받아 들여 elementGenerator[ ]에 저장한다.

AutoElement 클래스의 Source 코드는 코드 Head의 속성을 초기화하고, *AutoElement* 객체를 하나 생성하여 elementGenerator[ ]에 nullElement 생성함수를 대입하는 초기화를 수행한다. 이 초기화는 Main() 프로그램이 시작되기 전에 수행된다.

설명이 생략된 기타 함수 및 맴버변수는 자동설치와는 직접적인 관련이 없는 것으로 요소의 특성을 표현하기 위한 것이다.

여기서 Element 클래스는 모든 요소의 부모 클래스로 사용되는 추상 클래스로 대부분의 요소 속성이 여기에 정의되어 있고, 요소 강성 행렬 계산에 필요한 다양한 공용함수가 정의되어있다.

```
//----------------------------------------Head File----------------------------------------
#ifndef AUTOELEMENT
#define AUTOELEMENT
#include "Matrix.h"
#include "element.h"
class NullEl:public Element
{
    public:
        void printElementInform(){Log::outMessage("Null Element");}
                NullEl(){};
               ~NullEl(){};
};
#define MAXCOUNT 100
typedef Element* (* AutoEl )();
class AutoElement{
    public:
        static short maxCount;
        static AutoEl elementGenerator[MAXCOUNT];
        static short convertTable[MAXCOUNT];
        AutoElement(AutoEl name, int i){if(i>MAXCOUNT)Log::message("Error: Exceed
                                                MAXCOUNT",i,"stop");
            if(elementGenerator[i]!=nullEl_loader)Log::outMessage("Error Duplicated element
                                                        assigned at",i, "stop");
            elementGenerator[i]=name;
        }
        AutoElement(){setNullElement();}
       ~AutoElement(){}
        static short nodeInfo[MAXCOUNT][2]; // Basic inform.: Number of nodes, dof
            static short freedomInfo[MAXCOUNT][3];
            static char character[MAXCOUNT];
        static void setNullElement(){
                        for(int i=0;i<MAXCOUNT;i++)elementGenerator[i]=nullEl_loader;}
        static Element* nullEl_loader(){return (Element*)new NullEl();}
};
#endif
//-----------------------------------Source File-----------------------------------------
#include "autoelement.h"
short AutoElement::maxCount=MAXCOUNT;
short AutoElement::convertTable[MAXCOUNT];
short AutoElement::nodeInfo[MAXCOUNT][2];
short AutoElement::freedomInfo[MAXCOUNT][3];
AutoEl AutoElement::elementGenerator[MAXCOUNT];
char AutoElement::character[MAXCOUNT];
static AutoElement autoelement;
```

List. 2 AutoElement 클래스

4.2 요소 클래스

List 3에 예로 든 봉요소 경우를 요소 생성 템플레이트로 하여 다른 요소를 생성한다. List에서 굵게 표시된 부분만 다시 추가하려는 요소에 맞추어 다시 작성하고 강성 행렬 계산 코드를 추가하면 된다. 이 요소를 자동으로 설치하는 코드는 List의 맨 마지막 2행으로, main() 프로그램이 수행되기 전에 실행되는 코드이다. 좀 더 자세히 보면, 이 요소의 새로운 객체를 생성하는 정적함수 *Bar_loader()* 가 정의되어 있다. 또 AutoElement 객체인

Bar_o를 정적으로 생성하고 매개 변수로 요소 생성함수 *Bar_loader* 사용하면 이 함수 이름이 *AutoElement* 클래스의 elementGenerator[ ]에 저장된다. 이로서 기존의 코드에 요소를 추가하는 코드 실행이 완료된다. 이 요소를 컴파일하여 기존의 코드에 링크하는 것으로 추가가 완료된다.

4.3 요소의 생성과 사용

요소의 생성은 입력 자료에서 요소의 형태를 나타내는 숫자를 매개로 하여 다음과 같은 코드로 생성한다.

m->element[eCOUNT]=AutoElement::
elementGenerator[eTY]();

여기서 eCOUNT는 요소의 일련번호, eTY는 요소 고유번호임.
입력에 따라 ***eTY*** 를 바꾸어 사용하면 모든 요소를 위의 하나의 코드로 생성 가능하기 때문에 요소의 추가에 또는 종류에 따라 코드를

```
#if!defined BAREL
#define  BAREL
#include "autoelement.h"
#include "matrix.h"
#include "log.h"
#define IDNO_BarEl 6
class BarEl:Element
{
    private:
        static short ID;
    public:
        void kMatrix(Matrix &EK, Matrix &MS, Matrix &EF, SolverProtocol &protocol);
        void printElementInform(){Log::outMessage(ID," : Bar Element");}
            BarEl(){
                if(ID==-IDNO_BarEl){ID=IDNO_BarEl;
                AutoElement::nodeInfo[ID][0]=2;
                AutoElement::nodeInfo[ID][1]=2;
                AutoElement::freedomInfo[ID][0]=1;
                AutoElement::freedomInfo[ID][1]=2;
                AutoElement::freedomInfo[ID][2]=3;
                AutoElement::character[ID]='N';
            };
            ~BarEl(){};
};
#endif
// 1d-3d
// you can fine result in rs->disp(freedom, load case)
#include "barEl.h"
#include "ResultInterface.h"
short BarEl::ID=-IDNO_BarEl;
void BarEl::kMatrix(Matrix &EK, Matrix &MS, Matrix &EF, SolverProtocol &protocol)
{
    --------강성 행렬, 질량 행열 등의 코드 추가----------
}
static Element* Bar_loader(){return (Element*)new BarEl();}
static AutoElement Bar_o(Bar_loader,IDNO_BarEl);
```

List. 3 Bar 요소 코드

바꿀 필요가 없다. 이것은 부모 클래스 객체가 자식 클래스를 객체를 참조할 수 있기 때문에 가능한 기법이다.

Solver에서는 다음과 같이 하나의 코드로 각 요소의 강성행렬을 구한다.

*if(el[onProcess]->gtype>=0)el[onProcess]->**kMatrix(EK,MS,EF,protocol);***

여기서 onProcess는 현재 계산 중인 요소 일련 번호, el[]은 요소객체 주소임.

5. 객체간 자료 전달 방법

모델 객체와 Solver 객체를 분리하여 서로 독립성을 높였기 때문에 객체간 자료전달을 방법을 고안해야한다. 자료전달 방법은 함수의 매개변수로 하는 방법이 있지만 프로그램의 관리 및 구조화의 견지에서 보면 바람직하지 못하다. 따라서 여기서는 객체의 주소를 관리하는 클래스를 별도로 만들어 객체를 다른 객체에 전달한다.

Model 모듈에서 생성된 객체의 주소를 관리하는 클래스를 만드는 데 참조할 객체와 동일한 형의 포인터 참조자를 멤버변수로 하고 이 참조자에 객체의 주소를 할당한다. 객체의 자료가 필요하면 이 주소관리 클래스를 상속받으면 원하는 객체를 참조할 수 있다. 주소 참조자는 정적 맴버변수로 되어 있기 때문에 아무리 많이 상속되어도 유일하게 존재하며, 상속하지 않고 헤드만 포함하여도 객체를 참조할 수 있다(정적이기 때문에 클래스 이름으로 접근 가능). 이러한 주소관리 클래스로는

. *AddressInterfaceElement:* 요소에 필요한 객체의 주소를 포함
. *AddressInterfacerSolver:* 해석를 구할 때 필요한 객체 주소
. *CoordinateSystemAddress:* 좌표계 객체 주소
. *ResultInterface* : 결과 객체 주소
. *JobsAddress:* 해석객체 주소등이 있다.

6. 기타 기능

본 프로그램은 수퍼요소를 만들고, 수퍼요소를 포함한 해석이라는 옵션으로 수퍼요소를 간편하게 사용할 수 있게 하였으며, 수퍼요소 생성을 위한 입력자료와 이용하기 위한 입력자료는 같은 파일을 사용하도록 하여 번거로움을 피했다. 또한 직접요소라는 요소를 개발하여 요소의 강성이 직접 메트릭스 형태로 주어지는 요소를 생성하고 보통 요소처럼 사용하는 기능이 있다. 많이 사용되는 구조 부분을 직접요소로 만들어 두면 여러 해석에 공통으로 사용이 가능하다.

절점 커플링 기능이 설치되어 있는데, 절점의 결합, 한 절점의 자유도를 부분적으로 해제하는 기능을 갖고 있으며, 국부좌표계 사용이 가능하다.

7. 입력자료

입력자료는 Tag를 이용하기 때문에 자료의 입력 순서에 상관이 없다. 모든 자료를 문자로 받아들이고 분해하여 원하는 자료형식으로 변환하기 때문에 문자, 숫자 자료를 혼합하여 작성할 수 있다. 여기서 인터페이스에 등록된 Tag 외는 모두 무시한다.

새로운 종류의 입력자료 형식을 추가하기 위해서는 자료를 구분하는 Tag를 인터페이스

에 다음과 같이 등록하고, 그 자료를 읽는 인터페이스 멤버 함수를 만들면 된다.

else if(dcv.stringCmpB("MP")) readMaterial();

그리고 *DefaultInterface* 에 설치된 "LIB_INFORM", "JOB_INFOM" 이라는 Tag를 사용하면 현재 설치되어 있는 요소 및 해석 종류를 조회할 수 있다. 현재 설치된 요소는 17개이다.

추후 XML 형식으로 Tag를 만들어 모든 응용프로그램에 공용으로 사용할 Tag를 고안할 예정이다.

8. 프로그램 실행 구조

List 4는 main() 프로그램의 일부로 해석의 수행 먼저 입력자료 파일명을 *FileManager* 에 등록하는 것으로 시작된다. Log 객체를 생성하여 log 파일과 출력 파일을 생성하고 수치 적분에 필요한 자료를 활성화시킨다.

다음은 모델과 인터페이스 객체를 생성한다. 이 인터페이스 객체를 생성할 때 모델 객체를 매개로 변수로 전달한다. 인터페이스 객체인 *readModel*의 맴버 함수 *doReadingWork();* 로 입력자료를 입력한다. 해석 결과를 관리하는 객체를 생성하고 그 주소를 주소 클래스에 등록한다. 끝으로 해석 종류를 나타내는 *js* 객체의 *doJob()* 함수를 실행하면 해석이 수행된다. 이 *js*가 참조하는 해석객체는 모델 객체에서 생성되어 *JobsAddress*에 등록된다.

9. 결 언

이 프로그램은 다른 유한요소 프로그램 개발 시 개발 프레임으로 사용하기 위하여 개발된 것으로 확장성을 최우선으로 하였다. 앞으로 Solver 모듈을 확장하고 모듈간의 독립성을 더욱 높이는 방향으로 개선할 예정이다. 다음은 이 연구의 결과를 요약한 것이다.

- 객체지향으로 설계된 유한요소 프로그램으로 총 65개의 클래스로 이루어져 있고 모든 자료와 함수는 클래스의 맴버로 되어 있음
- 기능별 모듈화를 통하여 프로그램 개발 및 보수/유지의 독립성을 높이고 모듈간의 자

```
FileManager::inputFileName(args);

Log log;  // start Log and out file, the name is take after args
GaussIntegrationBase gauss; //  loading integration information

Model *model;     //  FEM model set define & creation
model=new Model();
Interface *readModel;  //  FEM Model data input
readModel=AutoInterface::interfaceGenerator[0](model);
readModel->doReadingWork();
model->dumpModelToFile( );// this is for model check

Results *solution;  //  setting result container
solution=new Results();
ResultInterface::rs=solution;

JobsAddress::js->doJob();  //do job
```

List. 5 main 프로그램

료전달을 간결하게 하는 기법 개발 적용
- 클래스 자동 추가 기법을 개발하여 프로그램 확장성을 높였고, 인하우스 프로그램의 최대약점인 보수/유지가 어려운 점을 객체지향과 설계, 모듈화, UML로 표현된 설계도 등으로 극복
- 기타 연산 효율의 개선, 유한요소 모델의 크기와 상관없이 해석을 수행할 수 있도록 하였음

참고문헌

1. Bruce W. R. Forde, Richard O. Foschi and Siegfried F. Stiemer, Object-oriented finite element analysis , *Computer & Structure Vol. 34*, pp.355-401, 1990.
2. G.R.Miller, An object-oriented approach to structural analysis and design, *Computer & Structure Vol. 40, No.1*, pp.75-82, 1991.
3. S.P. Scholz, Elements of an object-oriented FEM++ program in C++ , *Computer & Structure Vol. 43*, No.3, pp.517-529, 1992.
4. Thomas Zimmermann, Yves Dubois-Pelerin, and Patricia Bomme, "Object-oriented finite element programming I. Governing principles", *Comput. Methods Appl. Mech. Engrg., Vol. 98*, pp.291-303, 1992.
5. Yves Dubois-Pelerin, Thomas Zimmermann, "Object-oriented finite element programming III. An Efficient implementation in C++", *Comput. Methods Appl. Mech. Engrg., Vol. 108*, pp.165-183, 1993.
6. J. Besson, R. Foerch, "Large scale object-oriented finite element code design", *Comput. Methods Appl. Mech. Engrg., Vol. 142*, pp.165-187, 1997.
7. G.C. Archer, G. Fenvaes, and C. Thewalt, "A new object-oriented finite element analysis program architecture", *Compute and Structures, Vol. 70*, pp.63-75, 1999.
8. Lichao Yu, Ashok V. Kumar, "An object-oriented modular framework for implementing the finite element method", *Compute and Structures, Vol. 79*, pp.919-928, 2001.
9. S. Modak, E.D. Sotelino, "An object-oriented programming framework for implementing the finite element method for the parallel dynamic analysis of structures", *Compute and Structures, Vol. 80*, pp.77-84, 2002.
10. Rainer Niekamp, Erwin Stein, "An object-oriented approach for parallel two-and-three dimensional adaptive finite element computations", *Compute and Structures, Vol. 80*, pp.317-328, 2002.