Homework 2 - Part A

Programming in Science and Engineering

Nino Guzmán: *ninogdo@ciencias.unam.mx*

May 2020

1 Introduction

The object-oriented programming (OOP) is useful for the implementation of large and complex programs, and it allow us to extend the program in a natural and easy way. Moreover, it maximize the reuse of the code and it has a nice integration between theory and implementation.

In this work we are going to give the general structure of a FEM program that will be implemented afterwards in C++ language. We are going to discuss the general features of this approach and compare it with the implementation done in the previous assignments.

Let us recall the some definitions that will clarify the proposed structure:

Class: It is a user-defined data type, which holds its own *data members* and *member functions*, which can be accessed and used by creating an instance of that class.

Object: It is an instance of a class. When this happens then memory is allocated.

The implementation is going to exploid the features of classes in the OOP:

1. Encapsulation: It is used to show only the essential information about the data, hiding details or implementation using abstraction.
2. Polymorphism: It allows to group and use different objects from derived classes and give them same interface.
3. Inheritance: It refers to new classes that inherits methods and data templates from other classes.

2 List of classes

The classes proposed afterwards can be divided into three main groups, depending on their main objectives, that are: *FEM classes*, *Numerical classes* and *Analysis classes*, similar to what it is proposed in [1].

2.1 FEM classes

The list of classes proposed in this section are basically the same that are found in the literarute. The **Node** class will represent a discrete point in the space with the number of degrees of freedom (**nDOF**) as the main attribute along their coordinates. In order to take into account the dimension of the problem we could use the constructors of the **Node** class.

Class Node	
Attributes	Methods
• **nDOF** • nodes coordinates	• getNodeCoordinates() • getLocalNumbering() • getnDOF() • evaluateVariable() • fixValue()

The **Element** class will performe basically the main computations of the matrices derived from the discretization. For this we present only the method *computeMatrix()* but we must keep in mind that this can be done to any kind of matrix (stiffness, mass, damping ...). This class will play the role of the reference element so it is going to be related with the **Node**, **Shape**, **IntegrationPoint**, **Material** and **Domain** classes.

Class Element	
Attributes	Methods
• Type of element • Dimension • Numbering • Interpolation • Number of nodes • Number of DOF	• getNodes() • getNodesCoordinates() • getnDOF() • computeMatrix() • computeSourceterm() • computeBoundaryConditions()

The **Shape** will contain the geometric and interpolation aspects of the element. It can be defined in an abstract way such that their derived classes contain the information about the dimension (1D,2D, 3D), the interpolation order, the basis functions, their derivatives and the Jacobian.

Class Shape	
Attributes	Methods
• Dimension • Interpolation order • Basis functions • Jacobian • Number of integration points	• evaluateShapeFunctions() • evaluateDerivatives() • computeJacobian()

The **IntegrationPoint** class holds the parametric coordinates and the corresponding weights in order to compute the numerical integration.

Class IntegrationPoint	
Attributes	Methods
• Parametric coordinates • Weights	• computeQuadrature() • evaluteAtIP()

The **Material** class will contain the information about some properties of the material that are going to be evaluated at the nodes.

Class Material	
Attributes	Methods
• Property • Nodes • Number of DOF	• evaluateProperty()

The **BoundaryConditions** are imposed at the nodes in some specific part of the domain. The derived classes will carry the methods depending on the type of boundary conditions we have. Two main BC are consider **Dirichlet**, **Neumann** but we can extend it as needed. For instance add **Robin** conditions.

Class BoundaryConditions	
Attributes	Methods
• Type of boundary conditions	• getBC()

Class Dirichlet	
Attributes	Methods
• Nodes • Number of DOF	• setValues()

Class Neumann	
Attributes	Methods
• Nodes • Number of DOF	• setRelation() • computeContributionMatrix() • computeContributionSourceterm()

The **Domain** class will contain the rest of the FEM classes. We can acces to the component of the **Domain** through their methods.

Class Domain	
Atributes	Methods
• Number of Elements • Number of Nodes in elements • Element numbering • Nodes numbering • Physical coordinates	• addElement() • removeElement() • getNodes() • getElement() • getBoundaryNodes()

2.2 Analysis classes

The **Analysis** classes will perform a similar analysis of what is done in the postprocess in the previous assignments. It will compute the convergence of the method.It can also perform the smoothing.

Class Error	
Atributes	Methods
• Numerical solution • Analytical solution • Number of DOF	• computeError()

2.3 Numerical classes

Finally the **Numerical** classes will perform the main computations in order to solve the linear (or non-linear) system of equations.

Class LinearSoE	
Atributes	Methods
• Matrices • Source term • Unknown	• solveSystem()

3 Diagrams

In this section we present the hierarchy diagram along the inheritance diagrams using the classes proposed before. We also proposed an alternative scheme adding some classes and their corresponding diagram.

Let us explain the type of relations between the classes:

1. **knows-a** This relation exhibits the knowledge between objects from different classes. It is represented by a line between two classes (–).
2. **is-a**. This relations exhibits inheritance, and it consists when a class (descendent) can be derived from another class (ancestor). It is represented by a line ending in a triangle (△). For instance, class **Element** is inherited by class **Domain**.
3. **has-a**. This relation represent an aggregation, and it consists when a class is made up from objects from other classes. It is represented by a line ending in a diamond symbol (◊).

In the first diagram (1) we show the inheritance and the hierarchy diagram using the classes previously mentioned.

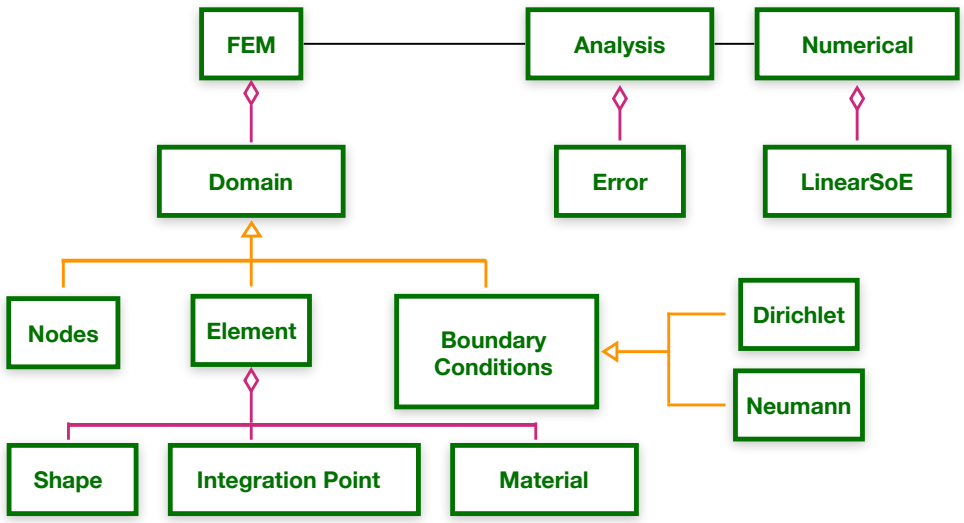

Figure 1: Hierarchy diagram showing inheritance and aggregation.

In the second diagram (2) we consider an additional **Model** class that will help pus to construct the model, as it is proposed in [2]. This class is defined in an abstract way and has no implementation. In this alternative we also consider additional boundary conditions and the **Smoothing** class within the **Analysis** classes.

Class Model	
Attributes	Methods
• Type of model	• buildModel()

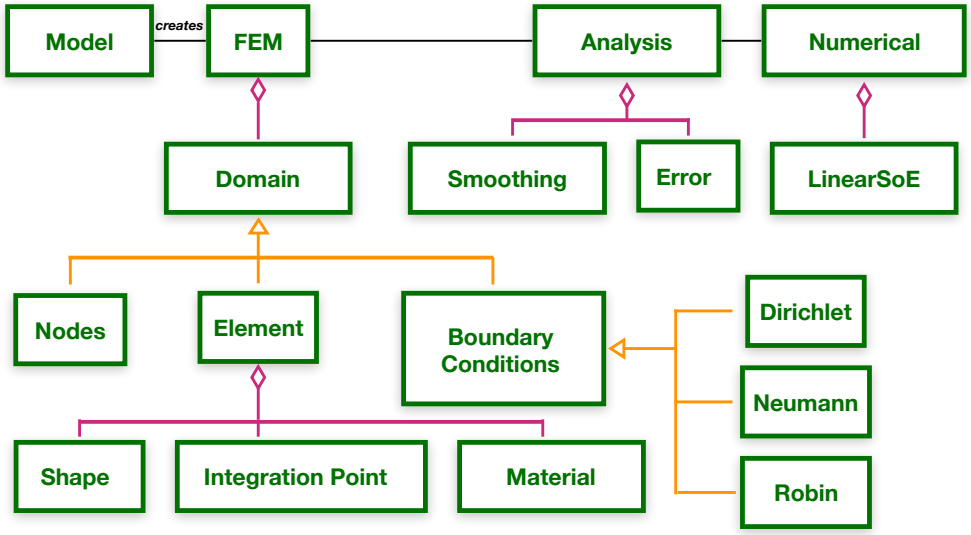

Figure 2: Hierarchy diagram showing inheritance and aggregation with additional classes

4 Conclusions

In this work we presented a design for finite element analysis using an object-oriented programming.

The proposed design is manly based on the one proposed [2], nevertheless we use some other classes mentioned in [1] and [3]. We think that a good property to this proposal is that it clarifies some aspects about the definition of the classes done in [2] and it is more flexible. The way in which is designed makes it easyly comparable with the previous implementations done in MATLAB.
We also consider a second alternative adding some classes that can be useful according to the considered problem.

Some drawback about our design are: first, we could loss some consistency when we consider additional classes, or when we modify the cited approaches, so we must keep in mind this issue when we perform the implementation. Second, as it is exposed in the literature, [1], [2], [3], [4] they consider some specific aspect about the physical model, mainly about mechanical (elastic) models. In our approach we lack this information. And finally we lack some specifications about the attributes and methods in the classes, but this will be done when we present the implementation details.

References

[1] F. McKenna, *Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel computing* , (1997)

[2] L. F. Martha, E. P. Junior *An Object-Oriented Framework for Finite Element Programming*, (2002)

[3] T. Zimmermann, Y. Dubois-Pelerin & P. Bomme *Object-oriented finite element programming: I. Governing principles*, (1991)

[4] T. Zimmermann, Y. Dubois-Pelerin *Object-oriented finite element programming: III. An efficient implementation in C++* , (1992)