Chapter 2

Background

The history of object-oriented design for finite element programs turns back to the early 90's or even earlier. In 1990, Forde, *et al.* [44] published one of the first detailed descriptions of applying object-oriented design to finite element programs. They used the essential components of finite element method to design the main objects. Their structure consists of `Node`, `Element`, `DispBC`, `ForcedBC`, `Material`, and `Dof` as finite element components and some additional objects like `Gauss`, and `ShapeFunction` for assisting in numerical analysis. They also introduced the element group and the list of nodes and elements for managing the mesh and constructing system. Their approach has been reused by several authors in organizing their object-oriented finite element program structures. This approach was focused on structural domain and the objects' interfaces reflect this dependency.

In those years other authors started to write about the object-oriented paradigm and its applications in finite element programming. Filho and Devloo [43] made an introduction to object oriented design applying it to element design. Mackie [64] gave another introduction to the object-oriented design of finite element programs by designing a brief hierarchial element. Later he published a more detailed structure for finite element program providing a data structure for entities and also introduced the use of iterators [65]. Pidaparti and Hudli [80] published a more detailed object oriented structure with objects for different algorithms in dynamic analysis of structures. Raphael and Krishnamoorthy [82] also made an introduction to object-oriented programming and provided a sophisticated hierarchy of elements for structural applications. They also designed some numerical expressions for handling the common numerical operations in finite element method. The common point of all these authors was their awareness about the advantages of object oriented programming with respect to traditional Fortran approaches and their intention to use these advantages in their finite element codes.

Miller [70, 71, 72] published an object-oriented structure for a nonlinear dynamic finite element program. He introduced a coordinate free approach to his design by defining the geometry class which handles all transformations from local to global coordinates. The principal objects in his design are `Dof`, `Joint`, and `Element`. `TimeDependentLoad` and `Constrain` are added to them in order to handle boundary conditions. He also defines a `Material` class with ability to handle both linear and nonlinear materials. The `Assemblage` class holds all these components and encapsulates the time history modeling of structure.

Zimmermann, *et al.* [106, 35, 34] have designed a structure for linear dynamic finite element analysis. Their structure consists of three categories of objects. First the finite element

method objects which are: `Node`, `Element`, `Load`, `LoadTimeFunction`, `Material`, `Dof`, `Domain`, and `LinearSolver`. The second category are some tools like `GaussPoint`, and `Polynomial`. The third category are the collection classes like: `Array`, `Matrix`, `String`, etc. They implemented first a prototype of this structure in *Smalltalk* and after that an efficient one in C++, which latter version provides a comparable performance respect to a Fortran code.

In their structure `Element` calculates the stiffness matrix K^e, the mass matrix M^e, and the load vector f^e in global coordinates. It also assembles its components in the global system of equations and update itself after solving. `Node` holds its position and manages dofs. It also computes and assembles its load vector and finally update the time dependent data after solving. `Dof` holds the unknown information and also its value. It stores also its position in the global system. and provides information about boundary conditions. `TimeStep` implements the time discretization. `Domain` is a general manager which manages the components like nodes and elements and also manages the solving process. It also provides the input-output features for reading the data and writing the results. Finally `LinearSystem` holds the system of equation components: left hand side, right hand side and solution. It also performs the equation numbering and implements the solver for solving the global system of equations.

They also developed a nonlinear extension to their structure which made them redefined some of their original classes like `Domain`, `Element`, `Material`, and some `LinearSystems` [69].

Lu, *et al.* [61, 62] presented a different structure in their finite element code FE++. Their structure consists of small number of finite element components like `Node`, `Element`, `Material`, and `Assemble` designed over a sophisticated numerical library. In their design the `Assemble` is the central object and not only implements the normal assembling procedure but also is in charge of coordinate transformation which in other approaches was one of the element's responsibilities. It also assigns the equation numbers. The `Element` is their extension point to new formulations. Their effort in implementing the numerical part lead to an object-oriented linear algebra library equivalent to LAPACK [15]. This library provides a high level interface using the object-oriented language features.

Archer, *et al.* [17, 18] presented another object-oriented structure for a finite element program dedicated to simulate linear and nonlinear, static and dynamic structures. They reviewed features provided by different other designs on that time and combined them in a new structure adding also some new concepts.

Their design consists of two level of abstractions. In the top level, the `Analysis` encapsulates the algorithms and the `Model` represents the finite element components. `Map` relates the dofs in the model, to unknowns in the analysis and removes the dependency between these objects. It also transforms the stiffness matrix from the element's local coordinate to the global one and calculates the responses. Additional to these three objects, different handlers are used to handle model dependent parts of algorithm. The `ReorderHandler` optimizes the order of the unknowns. The `MatrixHandler` provides different matrices and construct them over given model. Finally `ConstraintHandler` provides the initial mapping between the unknowns of analysis and the dofs in the model.

In another level there are different finite element components representing the model. `Node` encapsulates a usual finite element node which holds its position, and dofs. `ScalarDof` and `VectorDof` are derived from the `Dof` class and represent the different degree of freedom's types. `Element` uses the `ConstitutiveModel` and `ElementLoad` to calculate stiffness matrix K^e, mass matrix M^e, damp matrix C^e in local coordinate system. `LoadCase` consists of loads, prescribed displacements, and initial element state objects and calculates the load vector in the local coordinate system.

Cardona, *et al.* [25, 53, 54] developed the Object Oriented Finite Elements method Led by Interactive Executor (OOFELIE) [77]. They designed a high level language and also implemented

an interpreter to execute inputs given in that language. This approach enabled them to develop a very flexible tool to deal with different finite element problems. In their structure a `Domain` class holds data sets like: `Nodeset`, `Elemset`, `Fixaset`, `Loadset`, `Materset`, and `Dofset`. `Element` provides methods to calculate the stiffness matrix K^e, the mass matrix M^e, etc. `Fixaset` and `Loadset` which hold `Fixation`s and `Load`s handl the boundary conditions and loads.

They used this flexible tool also for solving coupled problems where their high level language interpreting mechanism provides an extra flexibility in handling different algorithms in coupled problems. They also added `Partition` and `Connection` classes to their structure in order to increase the functionality of their code in handling and organizing data for coupled problems. `Partition` is defined to handle a part of domain and `Connection` provides the graph of degrees of freedom and also sorts them.

Touzani [96] developed the Object Finite Element Library (OFELI) [95]. This library has an intuitive structure based on finite element methodology and can be used for developing finite element programs in different fields like heat transfer, fluid flow, solid mechanics, and electromagnetic.

`Node`, `Element`, `Side`, `Material`, `Shape` and `Mesh` are the main components of its structure and different problem solver classes implement the algorithms. This library also provides different classes derived form `FEEqua` class which implement formulations in different fields of analysis. It uses a static `Material` class in which each parameter is stored as a member variable. The `Element` only provides the geometrical information and finite element implementation is encapsulated via `FEEqua` classes. The `Element` provides several features which make it useful for even complex formulations but keeping all these members data makes it too heavy for standard industrial implementations.

Bangerth, *et al.* [22, 19, 21] created a library for implementing adaptive meshing finite element solution of partial differential equations called Differential Equations Analysis Library (DEAL) II [20]. They were concerned with flexibility, efficiency and type-safety of the library and also wanted to implement a high level of abstraction. All these requirements made them to use advanced features of C++ language to achieve all their objectives together.

Their methodology is to solve a partial differential equation over a domain. `Triangulation`, `FiniteElement`, `DoFHandler`, `Quadrature`, `Mapping`, and `FEValues` are the main classes in this structure. `Triangulation` despite its name is a mesh generator which can create line segments, quadrilaterals and hexahedra depending on given dimensions as its template parameter. It also provides a regular refinement of cells and keeps the hierarchical history of mesh. The `FiniteElement` encapsulates the shape function space. It computes the shape function values for `FEValues`. `Quadrature` provides different orders of quadratures over cells. `Mapping` is in charge of the coordinate transformation. `DoFHandler` manages the layout of degrees of freedoms and also their numbering in a triangulation. `FEValues` encapsulates the formulation to be used over the domain. It uses `FiniteElement`, `Quadrature`, and `Mapping` to perform its calculation and provides the local components to be assembled into the global solver.

Extensive use of templates and other advanced features of C++ programming language increases the flexibility of this library without sacrificing its performance. They created abstract classes in order to handle uniformly geometries with different dimensions. In this way they let users create their formulation in a dimension independent way. Their approach also consists of implementing the formulation and algorithms and sometimes the model itself in C++. In this way the library configures itself better to the problem and gains better performance but reduces the flexibility of the program by requiring it to be used as a closed package. In their structure there is no trace of usual finite element components like node, element, condition, etc. This makes it less familiar for developers with usual finite element background.

Patzák, *et al.* [79] published an structure used in the Object Oriented Finite Element Modeling

(OOFEM) [78] program. In this structure `Domain` contains the complete problem description which can be divided into several `Engineering Model`s which prepare the system of equations for solving. `Numerical Method` encapsulates the solution algorithm. `Node`, `Element`, `DOF`, `Boundary condition`, `Initial condition`, and `Material` are other main object of this structure. This program is oriented to structural analysis.

2.1 Discussion

A large effort has been done to organize the finite element codes trying to increase their flexibility and reducing the maintenance cost. Two classes of designing finite element program can be traced in literature. One consists of using the finite element methodology for the design which leads to objects like element, node, mesh, etc. Another approach is to deal with partial differential equations which results in object functions working with matrices and vectors over domains.

The work of Zimmermann, *et al.* [106, 35, 34] is one of the classical approaches in designing the code structure considering finite element methodology. However there is no geometry in their design and new components like processes, command interpreter etc. are not addressed.

The effort of Miller, *et al.* in order to encapsulate the coordinate transformation in geometry is useful for relaxing the dependency of elemental formulation to the specific geometry.

Cardona, *et al.* [25, 53, 54] added an interpreter to manage the global flow of the code in a very flexible way. The interpreter was used for introducing new solving algorithms to the program without changing it internally. This code is also extended to solve coupled problems using the interpreter for introducing interaction algorithms which gives a great flexibility to it. The drawback of their approach is the implementation of a new interpreter with a newly defined language beside binding to an existing one. This implies the maintenance cost of the interpreter itself and prevent them from using other libraries which may have interfaces to chosen script languages.

Touzani [96] designed an structure for multi-disciplinary finite element programs. His design is clear and easy to understand but uses field specific interfaces for its component which are not uniform for all fields. This reduces the reusability of the algorithm from one field to the other.

The approach of Lu, *et al.* [61, 62] is on the line of designing a partial differential solver with emphasizes on numerical components. Archer, *et al.* [17, 18] extended this approach to a more flexible and extendible point and more recently Bangerth, *et al.* used the same approach in designing their code. However the structure results from this design can be unfamiliar to usual finite element programmers. For instance in the latter design there are no objects to represents nodes and elements, which are the usual components for finite element programmers.

In this work the standard finite element objects like nodes, elements, conditions, etc. are reused from previous designs but modified and adapted to the multi-disciplinary perspective. There are also some new components like model part, process, kernel and application are added to cover new concepts mainly arising in multi-disciplinary problems. A new variable base interface is designed providing a uniform interface between fields and increasing the reusability of the code. The idea of using an interpreter is applied by using an existing interpreter. A large effort is also done to design and implement a fast and very flexible data structure enable to store any type of data coming from different fields and guarantee the uniform transformation of data. An extendible IO is also created to complete the tools for dealing with the usual bottlenecks when working with multi-disciplinary problems.