Chapter 4

General Structure

In this chapter first the objectives and also the users considered in the design of Kratos are described then the methodology to design the structure is given.

4.1 Kratos' Requirements

Kratos is designed as a framework for building multi-disciplinary finite element programs [31]. Generality in design and implementation is the first requirement. Kratos has to provide the general tools necessary for finite element solution. It also has to remove many restrictions that exist in other codes in order to achieve enough flexibility to handle a wider variety of algorithms than before. For example, restrictions like using certain degrees of freedom (dof).

Kratos must provide a flexible structure in order to handle a wide variety of methods and algorithms. This flexibility has to be provided not only in its global layout and basic assumptions but also in its implementation details. Minimization of restrictive assumptions is necessary in order to let developers configure this library as they want at different levels.

Kratos as a library must provide a good level of reusability in its provided tools. The key point here is to help users develop easier and faster their own finite element code using generic components provided by Kratos, or even other applications.

Kratos has to be extendible, at different levels of implementation. It must be extendible to new formulations, algorithms, and concepts. Supporting a wide variety of problems that can be coupled in a multi-disciplinary problem requires very different formulations and algorithms to be implemented. These formulations and algorithms may also use new concepts and variables. So Kratos must provide an extendible design for all of its components in order to support new methods.

Another important requirements are good performance and memory efficiency. This features are necessary for enabling applications implemented using Kratos, to deal with industrial multi-disciplinary problems. These requirements are very important and are the reason for most of the restrictions in Kratos.

Finally it has to provide different levels of developers' contributions to the Kratos system, and match their requirements and difficulties in the way they extend it. Developers may want to just make a plug-in extension, create an application over it, or using IO scripts to make Kratos perform a certain algorithm. Kratos has to provide not only all these capabilities but also hide the unnecessary difficulties from each group of developers.

4.2 Users

One of the important factors in design is to determine who will work with the program, what are their needs and how the program can help them. In essence Kratos is defined to be used by three groups of users at different levels:

Finite Element Developers Kratos is defined to be used by finite element developers to implement a multi-disciplinary formulation easily. These developers, or users from Kratos point of view, are considered to be more expert in FEM, from the physical and mathematical points of view, than C++ programming. For this reason, Kratos has to provide their requirements without involving them in advanced programming concepts.

Application Developers Kratos can be used as a finite element engine for other applications. This ability favors another teams of developers to work with Kratos. These users are less interested in finite element programming and their programming knowledge may vary from very expert to higher than basic. They may use not only Kratos itself but also any other applications provided by finite element developers, or other application developers. Developers of optimization programs or design tools are the typical users of this kind.

Package Users Engineers and designers are other users of Kratos. They use the complete package of Kratos and its applications to model and solve their problem without getting involved in internal programming of this package. For these users Kratos has to provide a flexible external interface to enable them use different features of Kratos without changing its implementation.

Kratos has to provide a framework such that a team of developers with completely different fields of expertise as mentioned before, work on it in order to create multi-disciplinary finite element applications.

4.3 Object Oriented Design

History of object-oriented design for finite element programs turns back to early 90's, and even more. Before that, many large finite element programs were developed in modular ways. Industry demands for solving more complex problems from one side, and the problem of maintaining and extending the previous programs from the other side, has lead developers to target their design strategy towards an object-oriented one [44, 43, 64, 80, 82].

The main goal of an object-oriented structure is to split the whole problem into several objects and to define their interfaces. There are many possible ways to do this for each kind of problem we want to program and the functionality of the resultant structure depends largely on it. In the case of finite element problems there are also many approaches such as constructing objects based on partial differential equations solving methods [22] or in the finite element method itself [44, 106, 35, 34].

In Kratos we have chosen the second approach and have constructed our objects based on a finite element general methodology. This approach was selected because our goal was to create a finite element environment for multidisciplinary problems. Also our colleagues were, in general, more familiar with this methodology than with physical properties. In addition, this approach has given us the necessary generality mentioned above in the objectives of Kratos. Within this scope main objects are taken from various parts of the FEM structure. Then, some abstract objects are defined for implementation purposes. Finally their relation are defined and their responsibilities are balanced. Figure 4.1 shows the main classes in Kratos.

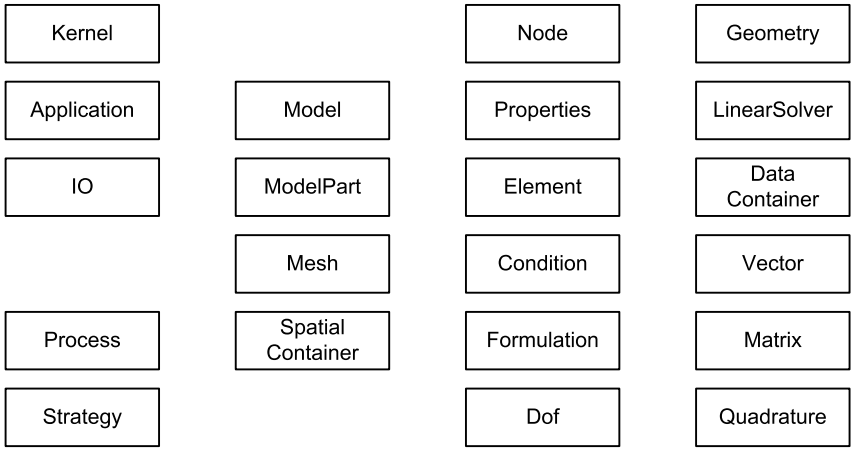

Figure 4.1: Main classes defined in Kratos.

`Vector`, `Matrix`, and `Quadrature` are designed by basic numerical concepts. `Node`, `Element`, `Condition`, and `Dof` are defined directly from finite element concepts. `Model`, `Mesh`, and `Properties` are coming from practical methodology used in finite element modeling completed by `ModelPart`, and `SpatialContainer`, for organizing better all data necessary for analysis. `IO`, `LinearSolver`, `Process`, and `Strategy` are representing the different steps of finite element program flow. and finally `Kernel` and `Application` are defined for library management and defining its interface.

These main objects are described below:

`Vector` Represents the algebraic vector and defines usual operators over vectors.

`Matrix` Encapsulate matrix and its operators. There are different matrix classes are necessary. The most typical ones are dense matrix and compressed row matrix.

`Quadrature` Implements the quadrature methods used in finite element method. For example the gaussian integration with different number of integration points.

`Geometry` Defines a geometry over a list of points or `Node`s and provides from its usual parameter like area or center point to shape functions and coordinate transformation routines.

`Node` `Node` is a point with additional facilities. Stores the nodal data, historical nodal data, and list of degrees of freedom. It provides also an interface to access all its data.

`Element` Encapsulates the elemental formulation in one objects and provides an interface for calculating the local matrices and vectors necessary for assembling the global system of equations. It holds its geometry that meanwhile is its array of `Node`s. Also stores the elemental data and interface to access it.

`Condition` Encapsulates data and operations necessary for calculating the local contributions of `Condition` in global system of equations. Neumann conditions are example of `Condition`s which can be encapsulated by derivatives of this class.

`Dof` Represents a degree of freedom (dof). It is a lightweight object which holds the its variable, like `TEMPERATURE`, its state of freedom, and a reference to its value in data structure. This

class enables the system to work with different set of dofs and also represents the Dirichlet condition assigned to each dof.

`Properties` Encapsulates data shared by different `Element`s or `Condition`s. It can stores any type of data and provide a variable base access to them.

`Model` Stores the whole model to be analyzed. All `Node`s, `Properties`, `Element`s, `Condition`s and solution data. It also provides and access interface to these data.

`ModelPart` Holds all data related to an arbitrary part of model. It stores all existing components and data like `Node`s, `Properties`, `Element`s, `Condition`s and solution data related to a part of model and provides interface to access them in different ways.

`Mesh` Holds `Node`s, `Properties`, `Element`s, `Condition`s and represents a part of model but without additional solution parameters. It provides access interface to its data.

`SpatialContainer` Containers associated with spacial search algorithms. This algorithms are useful for finding the nearest `Node` or `Element` to some point or other spacial searches. Quadtree and Octree are example of these containers.

`IO` Provides different implementation of input output procedures which can be used to read and write with different formats and characteristics.

`LinearSolver` Encapsulates the algorithms used for solving a linear system of equations. Different direct solvers and iterative solvers can be implemented in Kratos as a derivatives of this class.

`Strategy` Encapsulates the solving algorithm and general flow of a solving process. Strategy manages the building of equation system and then solve it using a linear solver and finally is in charge of updating the results in the data structure.

`Process` Is the extension point for adding new algorithms to Kratos. Mapping algorithms, Optimization procedures and many other type of algorithms can be implemented as a new process in Kratos.

`Kernel` Manages the whole Kratos by initializing different part of it and provides necessary interface to communicate with applications.

`Application` Provide all information necessary for adding an application to Kratos. A derived class from it is necessary to give kernel its required information like new `Variable`s, `Element`s, `Condition`s, etc.

The main intention here was to hide all difficult but common finite element implementations like data structure and IO programming from developers.

4.4 Multi-Layers Design

Kratos uses a *multi-layer* approach in its design. In this approach each object only interfaces with other objects in its layer or in layers below its layer. There are some other layering approaches that limited the interface between objects of two layers but in Kratos this limitation is not applied.

Layering reduces the dependency inside the program. It helps in the maintenance of the code and also helps developers in understanding the code and clarifies their tasks.

In designing the layers of the structure different users mentioned before are considered. The layering are done in a way that each user has to work in the less number of layers as possible. In this way the amount of the code to be known by each user is minimized and the chance of conflict between users in different categories is reduced. This layering also lets Kratos to tune the implementation difficulties needed for each layer to the knowledge of users working in it. For example the finite element layer uses only basic to average features of C++ programming but the main developer layer use advanced language features in order to provide the desirable performance.

Following the current design mentioned before, Kratos is organized in the following layers:

Basic Tools Layer Holds all basic tools used in Kratos. In this layer using advance techniques in C++ is essential in order to maximize the performance of these tools. This layer is designed to be implemented by an expert programmer and with less knowledge of FEM. This layer may also provides interfaces with other libraries to take benefit of existing work in area.

Base Finite Element Layer This layer holds the objects that are necessary to implement a finite element formulation. It also defines the structure to be extended for new formulations. This layer hides the difficult implementations of nodal and data structure and other common features from the finite element developers.

Finite Element Layer The extension layer for finite element developers. The finite element layer is restricted to use the basic and average features of language and uses the component base finite element layer and basic tools to optimize the performance without entering into optimization details.

Data Structure Layer Contains all objects organizing the data structure. This layer has no restriction in implementation. Advanced language features are used to maximize the flexibility of the data structure.

Base Algorithms Layer Provides the components building the extendible structure for algorithms. Generic algorithms can also be implemented here to help developer in their implementation by reusing them.

User's Algorithms Layer Another layer to be used by finite element programmers but at a higher level. This layer contains all classes implementing the different algorithms in Kratos. Implementation in this layer requires medium level of programming experience but a higher knowledge of program structure than the finite element layer.

Applications' Interface Layer This layer holds all objects that manage Kratos and its relation with other applications. Components in this layer are implemented using high level programming techniques in order to provide the required flexibility.

Applications Layer A simple layer which contains the interface of certain applications with Kratos.

Scripts Layer Holds a set of IO scripts which can be used to implement different algorithms from outside Kratos. Package users can use modules in this layer or create their own extension without having knowledge of C++ programming or the internal structure of Kratos. Via this layer they can activate and deactivate certain functionalities or implement a new global algorithm without entering into Kratos implementation details.

Figure 4.2 shows the multi-layer nature of Kratos.

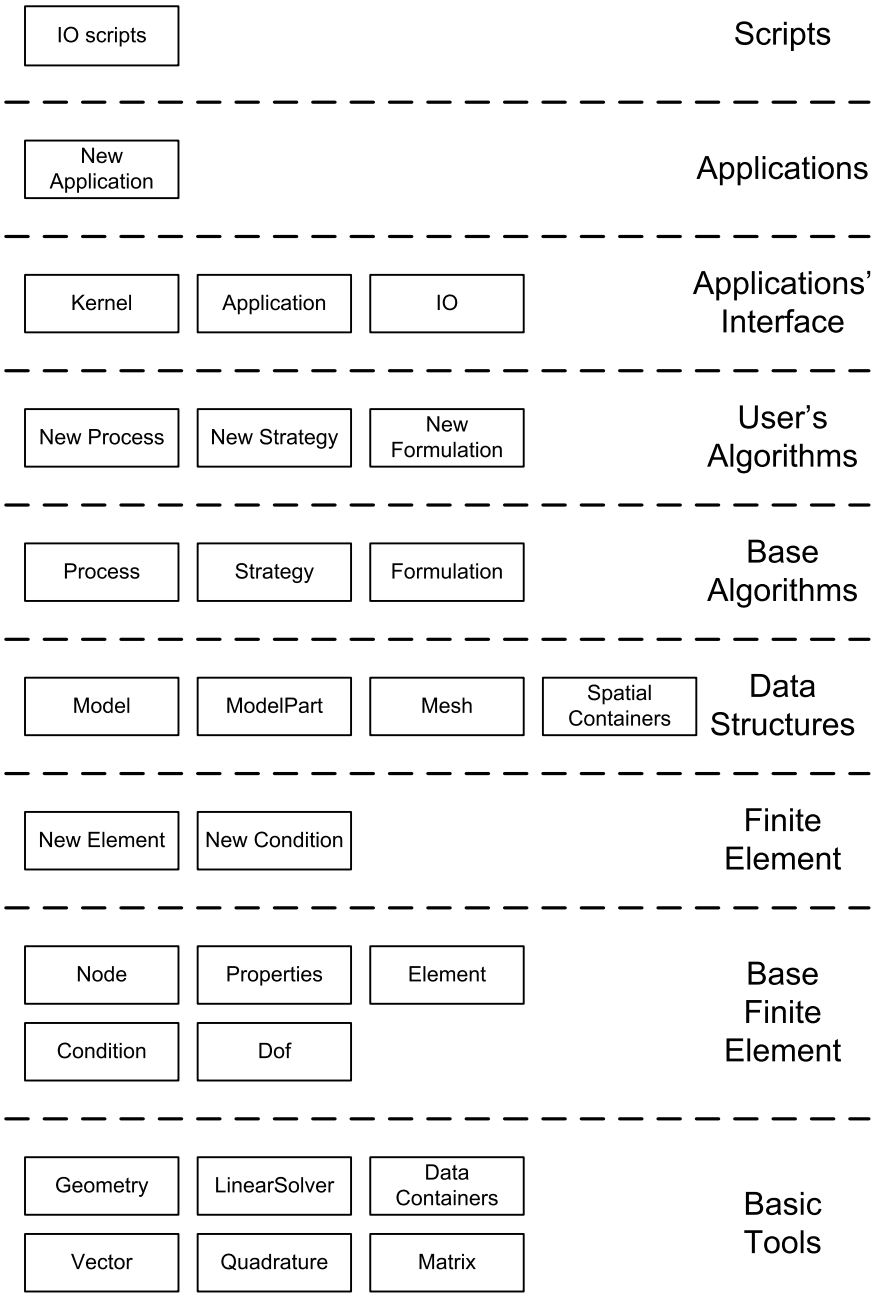

Figure 4.2: Dividing the structure into layers reduces the dependency.

4.5 Kernel and Applications

In the first implementation of Kratos all applications were implemented in Kratos and also were compiled together. This approach at that time produced several conflicts between applications and was requiring many unnecessary recompiling of the code for changes in other applications. All these problems lead to a change in the strategy and to separating each application not only from others but also from Kratos itself.

In the current structure of Kratos each application is created and compiled separately and just uses a standard interface to communicate with the kernel of Kratos. In this way the conflicts are reduced and the compilation time is also minimized. The `Application` class provides the interface for introducing an application to the kernel of Kratos. `Kernel` uses the information given by `Application` through this interface to mange its components, configure different part of Kratos, and synchronize the application with other ones. The `Application` class is very simple and consists of registering the new components like: `Variable`s, `Element`s, `Condition`s, etc. defined in application. The following code shows a typical application class definition:

```
// Variables definition
KRATOS_DEFINE_VARIABLE(int, NEW_INTEGER_VARIABLE )
KRATOS_DEFINE_3D_VARIABLE_WITH_COMPONENTS(NEW_3D_VARIABLE);
KRATOS_DEFINE_VARIABLE(Matrix, NEW_MATRIX_VARIABLE)

class KratosNewApplication : public KratosApplication
{
public:
  virtual void Register();

private:

  static const NewElementType    msNewElement;
  static const NewConditionType  msNewCondition;
};
```

Here `Application` defines its new components and now its time to implement the `Register` method:

```
// Creating variables
KRATOS_CREATE_VARIABLE(NEW_INTEGER_VARIABLE )
KRATOS_CREATE_3D_VARIABLE_WITH_COMPONENT(NEW_3D_VARIABLE);
KRATOS_CREATE_VARIABLE(NEW_MATRIX_VARIABLE)

void KratosR1StructuralApplication::Register()
{
  // calling base class register to register Kratos components
  KratosApplication::Register();

  // registering variables in Kratos.
  KRATOS_REGISTER_VARIABLE(NEW_INTEGER_VARIABLE)
  KRATOS_REGISTER_3D_VARIABLE_WITH_COMPONENTS(NEW_3D_VARIABLE);
  KRATOS_REGISTER_VARIABLE(NEW_MATRIX_VARIABLE)

  KRATOS_REGISTER_ELEMENT("MyElement", msNewElement);
  KRATOS_REGISTER_CONDITION("MyCondition", msNewCondition);
```

```
}
```

This interface enables Kratos to add all these `Variable`s, `Element`s, and `Condition`s in the list of components. Kratos also synchronizes the variables numbering between different applications. Adding new components to Kratos, enables IO to read and write them and also configures the data structure to hold these new variables.

In the next chapter the basic tools layer will be declared.