CMNE/CILAMCE 2007
Porto, 13 a 15 de Junho, 2007

AN OBJECT ORIENTED CLASS ORGANIZATION FOR DYNAMIC GEOMETRICALLY NON-LINEAR FEM ANALYSIS

Flavio T. Fonseca*, Roque L. S. Pitangueira

Departamento de Engenharia de Estruturas
Escola de Engenharia
Universidade Federal de Minas Gerais
Avenida do Contorno 842, 2º andar, 30110-060, Belo Horizonte, MG, Brasil
e-mail: flaviotf@dees.ufmg.br, web: http://www.dees.ufmg.br/insane

Keywords: Finite Element Method, Geometrically Non-Linear Analysis, Dynamic Analysis, Object-Oriented Paradigm.

Abstract. *This article discusses the object oriented project for a finite element method software expansion, which aims to add the capability of performing a dynamic geometrically non-linear analysis of structures. The article's main focus is on the new classes which will be implemented. A brief revision of the formulation to be implemented is made and the incremental-iterative procedures used for solving the problem equations are also discussed. Many UML (Unified Modeling Language) diagrams are shown, indicating how the new classes will interact with the old ones and how do they provide all the expected features.*

1 INTRODUCTION

The possibilities offered by technological resources in software development form a huge field for research in the numerical and computational methods applied to engineering area.

Mastering these resources and applying them in the progressive improvement of models demand a segmented computational environment, suitable to changes and whose complexity may be increased in a gradual manner, as it is proposed by INSANE (*Interactive Structural Analysis Environment*), a software developed in the Structural Engineering Department of Universidade Federal de Minas Gerais which is available at the project site: *http://www.dees.ufmg.br/insane*.

Basing in the implementation of finite element method structural models existent in the software, it will be possible to increase complexities from already fixed concepts, avoiding restarting the process in each improvement.

The INSANE's computational environment is formed by three great applications: pre-processor, processor and post-processor, all the three implemented in Java programming language. The pre and post-processor are interactive graphical applications with pre and post-processing tools for the various discrete models. The processor is the application which represents the system's numerical nucleus an is responsible for obtaining the results for different structural analysis discrete models.

In this paper, it will be discussed an expansion to the numerical nucleus, which will add the capability of performing a dynamic geometrically non-linear analysis of structures.

2 FEM DYNAMIC GEOMETRICALLY NON-LINEAR ANALYSIS

In a dynamic analysis, the finite element method (FEM) equilibrium equation to be solved is:

$$\boldsymbol{M}\ \ddot{\boldsymbol{d}} + \boldsymbol{C}\ \dot{\boldsymbol{d}} + \boldsymbol{K}\ \boldsymbol{d} = \boldsymbol{F} \tag{1}$$

in which $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{C}$ is the damping matrix, $\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{F}$ is the force vector and $\boldsymbol{d}$, $\dot{\boldsymbol{d}}$ and $\ddot{\boldsymbol{d}}$ are, respectively, the displacement vector and its first and second time derivatives.

The dynamic formulation used is the Newmark-β method [1], which can be summarized by the following formulae:

$$\widetilde{\boldsymbol{K}}\ \boldsymbol{\Delta d}^{(i)} = \boldsymbol{\Delta}\widetilde{\boldsymbol{F}}^{(i)} \tag{2}$$

$$\boldsymbol{\Delta}\dot{\boldsymbol{d}}^{(i)} = \frac{\gamma}{\beta\Delta t}\boldsymbol{\Delta d}^{(i)} - \boldsymbol{R}^{(i)} \tag{3}$$

$$\boldsymbol{\Delta}\ddot{\boldsymbol{d}}^{(i)} = \frac{1}{\beta(\Delta t)^2}\boldsymbol{\Delta d}^{(i)} - \boldsymbol{Q}^{(i)} \tag{4}$$

$$\widetilde{\boldsymbol{K}} = \boldsymbol{K} + \frac{1}{\beta(\Delta t)^2}\boldsymbol{M} + \frac{\gamma}{\beta\Delta t}\boldsymbol{C} \tag{5}$$

$$\boldsymbol{\Delta}\widetilde{\boldsymbol{F}}^{(i)} = \boldsymbol{\Delta F}^{(i)} + \boldsymbol{M}\boldsymbol{Q}^{(i)} + \boldsymbol{C}\boldsymbol{R}^{(i)} \tag{6}$$

$$\boldsymbol{Q}^{(i)} = \frac{1}{\beta\Delta t}\dot{\boldsymbol{d}}^{(i)} + \frac{1}{2\beta}\ddot{\boldsymbol{d}}^{(i)} \tag{7}$$

$$\boldsymbol{R}^{(i)} = \frac{\gamma}{\beta}\dot{\boldsymbol{d}}^{(i)} + \left(\frac{\gamma}{2\beta} - 1\right)\Delta t^{(i)}\ddot{\boldsymbol{d}}^{(i)} \tag{8}$$

in which Δt is a suitable time increment, γ and β are the Newmark parameters.

Assuming that $\gamma = \frac{1}{2}$ and $\beta = \frac{1}{4}$, the constant acceleration method is obtained. In the same way, if $\gamma = \frac{1}{2}$ and $\beta = \frac{1}{6}$, the linear acceleration method is obtained.

For the geometrically non-linear analysis, the modified Newton-Raphson method is adopted [2]. From this method, we have:

$$\boldsymbol{M}\ {}^{t+\Delta t}\ddot{\boldsymbol{d}}^{(i)} + \boldsymbol{C}\ {}^{t+\Delta t}\dot{\boldsymbol{d}}^{(i)} + {}^{t}\boldsymbol{K}\ \delta\boldsymbol{d}^{(i)} = {}^{t+\Delta t}\boldsymbol{R} - {}^{t+\Delta t}\boldsymbol{F}^{(i-1)} \tag{9}$$

$$\Delta\boldsymbol{d}^{(i)} = \Delta\boldsymbol{d}^{(i-1)} + \delta\boldsymbol{d}^{(i)} \tag{10}$$

where $\Delta\boldsymbol{d}^{(i)}$ is the incremental displacement vector in relation to the previous time step in iteration i and $\delta\boldsymbol{d}^{(i)}$ is the incremental displacement vector in relation to the iteration i. The left superscript represent the time step and the right superscript represents the iterative step.

The right-hand side of equation (9) is the unbalanced force vector and represents the error obtained in the relative iteration. An iterative process must be done, until this error is smaller than a pre-defined tolerance value.

Mixing the two formulations described above, we get the formulae for the dynamic geometrically non-linear analysis:

$${}^{t}\widetilde{\boldsymbol{K}}\ \delta\boldsymbol{d}^{(i)} = {}^{t+\Delta t}\widetilde{\boldsymbol{R}}^{(i-1)} \tag{11}$$

$${}^{t}\widetilde{\boldsymbol{K}} = {}^{t}\boldsymbol{K} + a_0\boldsymbol{M} + a_3\boldsymbol{C} \tag{12}$$

$${}^{t+\Delta t}\widetilde{\boldsymbol{R}} = {}^{t+\Delta t}\boldsymbol{R} - {}^{t+\Delta t}\boldsymbol{F}^{(i-1)} - \boldsymbol{M}\ {}^{t+\Delta t}\ddot{\boldsymbol{d}}^{(i-1)} - \boldsymbol{C}\ {}^{t+\Delta t}\dot{\boldsymbol{d}}^{(i-1)} \tag{13}$$

$${}^{t+\Delta t}\ddot{\boldsymbol{d}}^{(i-1)} = a_0\ \Delta\boldsymbol{d}^{(i-1)} - a_1\ {}^{t}\dot{\boldsymbol{d}} - a_2\ {}^{t}\ddot{\boldsymbol{d}} \tag{14}$$

$${}^{t+\Delta t}\dot{\boldsymbol{d}}^{(i-1)} = a_3\ \Delta\boldsymbol{d}^{(i-1)} + a_4\ {}^{t}\dot{\boldsymbol{d}} - a_5\ {}^{t}\ddot{\boldsymbol{d}} \tag{15}$$

$$^{t+\Delta t}\ddot{\boldsymbol{d}} = a_0\ {}^{t}\Delta\boldsymbol{d} - a_1\ {}^{t}\dot{\boldsymbol{d}} - a_2\ {}^{t}\ddot{\boldsymbol{d}} \tag{16}$$

$$^{t+\Delta t}\dot{\boldsymbol{d}} = {}^{t}\dot{\boldsymbol{d}} + a_6\ {}^{t}\ddot{\boldsymbol{d}} + a_7\ {}^{t+\Delta t}\ddot{\boldsymbol{d}} \tag{17}$$

$$^{t+\Delta t}\boldsymbol{d} = {}^{t}\boldsymbol{d} + {}^{t}\Delta\boldsymbol{d} \tag{18}$$

the constants used above are:

$$a_0 = \frac{1}{\beta\Delta t^2} \qquad a_1 = \frac{1}{\beta\Delta t} \qquad a_2 = \frac{1}{2\beta} - 1$$

$$a_3 = \frac{\gamma}{\beta\Delta t} \qquad a_4 = 1 - \frac{\gamma}{\beta} \qquad a_5 = \left(1 - \frac{\gamma}{2\beta}\right)\Delta t$$

$$a_6 = (1-\gamma)\,\Delta t \qquad a_7 = \gamma\Delta t$$

The algorithm in Figure 1 summarizes this process.

3 CURRENT OBJECT-ORIENTED PROJECT

As said before, INSANE is implemented in Java, an object oriented programming (OOP) language. The choice for an OOP language was made because of the various benefits brought by this conception, as the easiness for code expansion and maintenance. One of the main Java features is its portability. As a language that combines the compilation and interpretation process, it can be compiled in one operational system or computer and executed in a different one.

The persistence of data between the three segments of INSANE is done by binary files or XML (eXtensible Markup Language) files. The XML is a technique of creating structured data based in a text file. What differs it to other markup language, as HTML, is the fact that the markup rules are defined by the developer in a most suitable way. It is also adopted as the standard format for exchanging data via internet, so in the future it will be possible to make INSANE a web based software.

The INSANE actual implementation is the results of many recent works: [3], [4], [5], [6], [7] and [8]. The numerical nucleus is formed by interfaces which represent the various abstractions of one numerical resolution for discrete models, each one with its own class hierarchy responsible for making its job in processing. Actually, it is organized in function of the interfaces `Assembler`, `Solution`, `Model` and `Persistence`, as show in Figure 2.

The `Assembler` interface is responsible for mounting the second order matrix system, with which is possible to represent many types of discrete problems (Eq. 19).

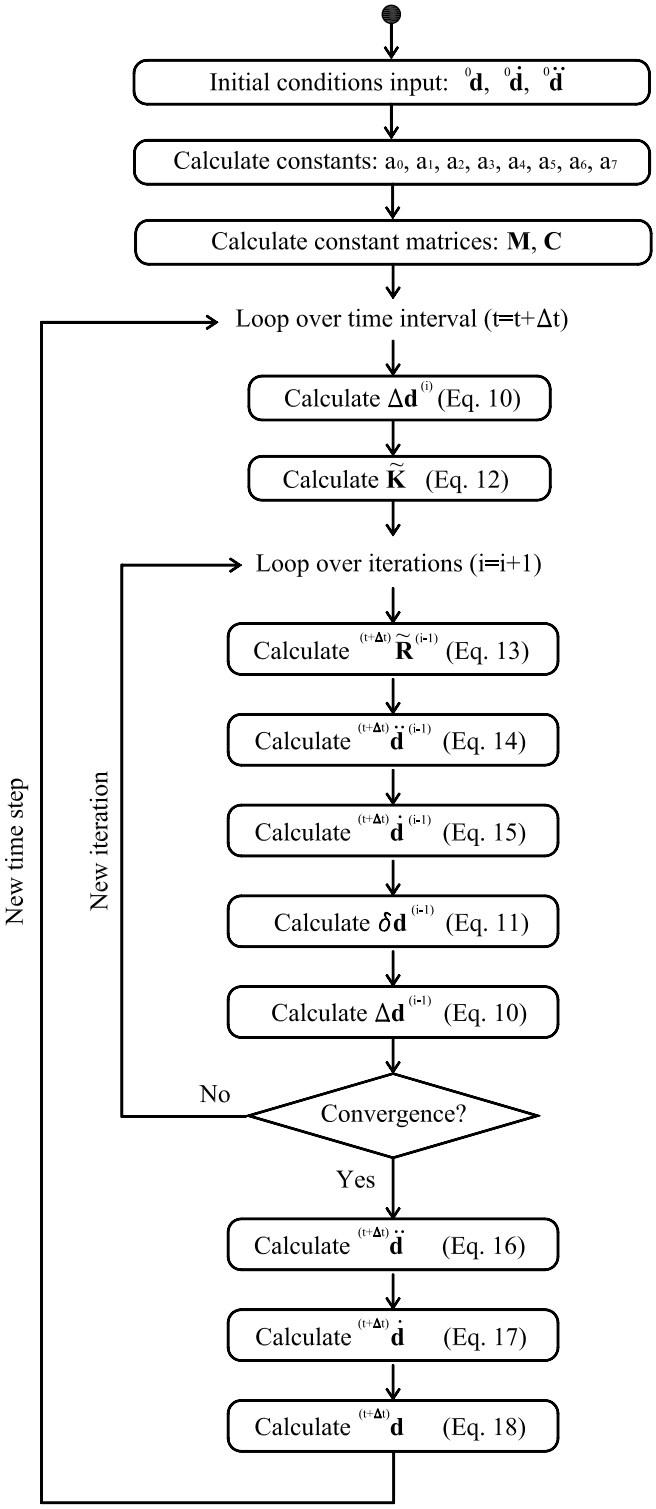

Figure 1: Activity diagram for dynamic non-linear analysis algorithm.

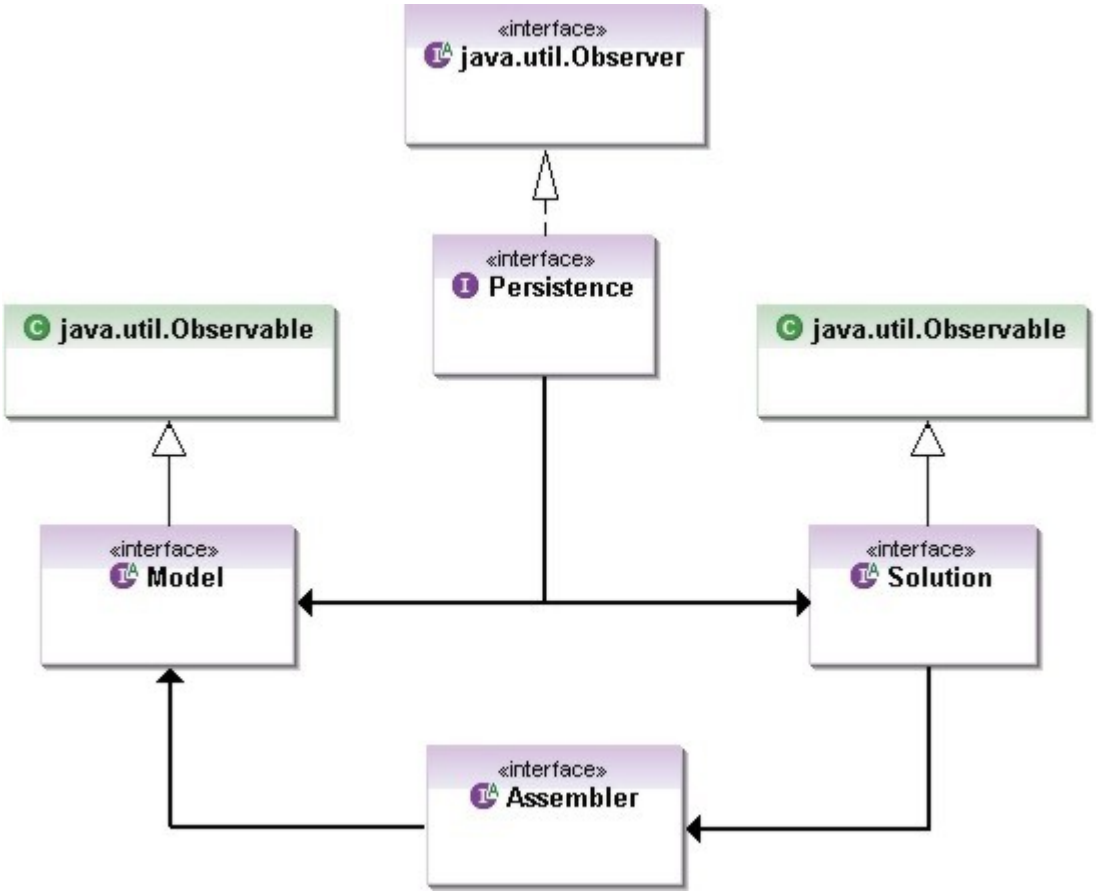

Figure 2: Organization of INSANE's numerical nucleus.

$$\boldsymbol{A}\,\ddot{\boldsymbol{X}} + \boldsymbol{B}\,\dot{\boldsymbol{X}} + \boldsymbol{C}\,\boldsymbol{X} = \boldsymbol{R} - \boldsymbol{F} \tag{19}$$

The `Solution` interface is who starts the solution process, possessing the necessary resources to solve this matrix system, linear or non-linear.

`Model` contains the data relative to the discrete model to be analysed and provides `Assembler` all the information necessary to mount model's equation, which will be solved by `Solution`.

Both `Model` and `Solution` communicate themselves with the `Persistence` interface, which treats the input data and, principally, persists the output data to the other applications, whenever it observes a modification of the discrete model state.

This modification observation occurs according to the *Observer-Observable* design pattern, which is a change propagation mechanism. When an object said *observer* (which implements the interface `java.util.observer`) is created, it is add to an observers list of some objects said *observables* (which extends the class `java.util.observable`). When a modification in the state of any observed object occurs, the change propagation mech-

anism is initialized, and the observer objects are notified to update themselves. This process guarantees the consistence and communication between the observer component (`Persistence`) and the observed components (`Solution` and `Model`).

It will be presented the main interfaces which compose the INSANE numerical nucleus. To facilitate understanding, many UML (Unified Modelling Language) diagrams will be shown.

3.1 Assembler Interface

The `Assembler` interface (Fig. 3) has the necessary methods to mount the model matrices and vectors according to equation (19). It is implemented by the class `FemAssembler`, which is appropriated to the various types of problems which can be modelled by the finite element method. The `FemAssembler` class has as an attribute one object of the type `Model`, which is the finite element model for which the equation must be mounted.

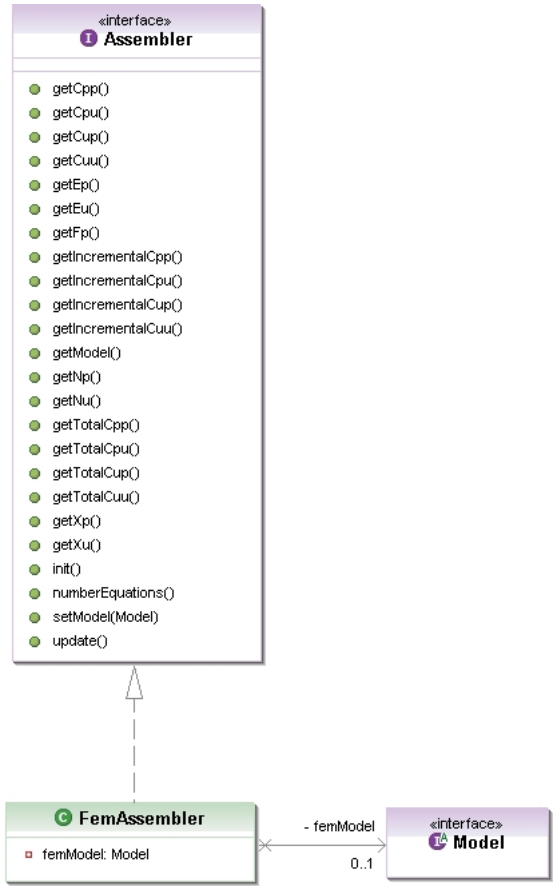

Figure 3: Class diagram for `Assembler`.

In the Figure 3, some methods of `Assembler` interface are shown. When called, these

methods are capable of providing any part of the matrix system. It is important to cite the method *numberEquations()*, which numbers the model's equations, organizing them according to the criterion of unknown or prescribed values. In this way, `Assembler` is capable of mount, for example, the reduced stiffness matrix. The methods *init()* and *update()* are related to the non-linear analysis process.

3.2 Solution Interface

Once the problem equation is mounted, it is a job for `Solution` interface (Fig. 4) to solve it. The classes which implements it are `SteadyState`, responsible for solving linear problems, and `EquilibriumPath`, responsible for solving non-linear problems through a incremental iterative process.

One object `SteadyState` has one object of the type Assembler and one object of the type `LinearEquationSystem`, which is responsible for obtaining the solution of a linear algebraic equations system, as the one mounted for the linear static solution (Fig. 4).

The class `EquilibriumPath` has an object `Step`, which has the methods for solving the incremental iterative process, and a list of `IterativeStrategy`, informed by the user, which defines the control methods to be used for obtaining the equilibrium paths. `EquilibriumPath` does not has an object of the type `Assembler`, but has the method *setAssembler(*`Assembler`*)* which sets to its object `Step` an object `Assembler` for which the incremental iterative method must be solved. It has also the method *execute()* which executes each step of the process. `EquilibriumPath` extends the class `Observable`, because it is observed by `Persistence`, and implements the class `Observer`, because it observes the state variations of its object `Step`.

Figure 5 shows the class diagram for `Step` interface, implemented by the class `StandardNewtonRaphson`, in which the standard Newton-Raphson method is used during the incremental iterative process of the non-linear solution.

`StandardNewtonRaphson` has all the attributes necessary to obtain the convergence in the step, emphasizing one object `Assembler`, capable of informing the matrices and vector of the equation to be solved in each iteration of the step; one object `LinearEquationSystem`, capable of solving the linear algebraic equations system of each iteration; and one object `IterativeStrategy`, which represents the current iterative strategy. `StandardNewtonRaphson` also extends the class `Observable`, being observed by `EquilibriumPath`.

The convergence is verified through the methods *getConvergence()* and *setConvergence()* of `StandardNewtonRaphson`. The method *setConvergence()* creates an object `Convergence`, which, using the method *checkConvergence()*, calculates the convergence based in force, displacement, or both, accordingly to what was informed by the user.

In the hierarchy of the interface `IterativeStrategy` (Fig. **??**) are the classes which represent the various control methods implemented. Each of these has the methods *getPredictor()* and *getCorrector()* which calculate the load factor for the first iteration and for the other ones, respectively. Except for `LoadControl`, all the other classes have one object of type `Step`, because they need it to get information about the previous step.

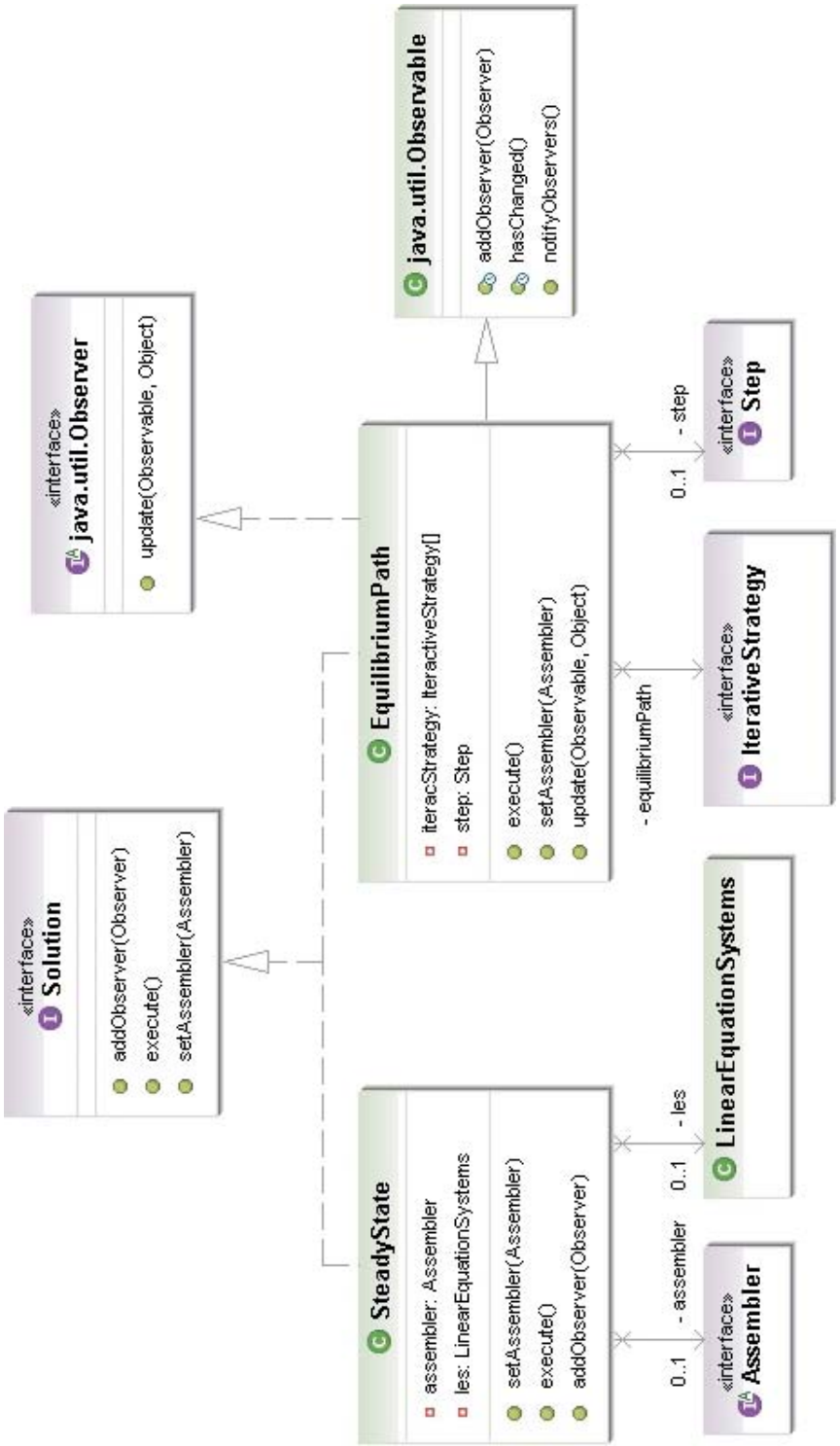

Figure 4: Class diagram for `Solution`.

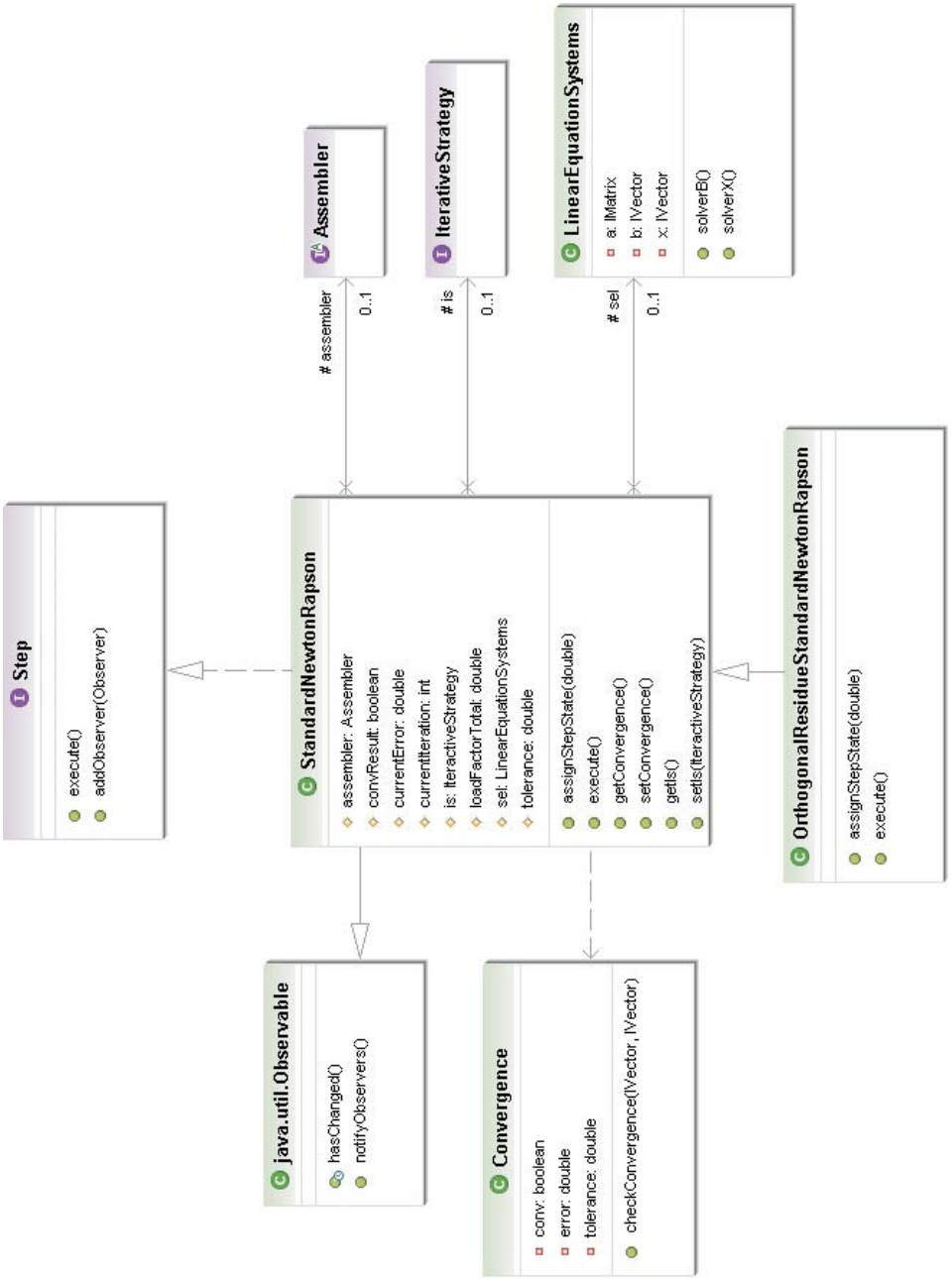

Figure 5: Class diagram for `Step`.

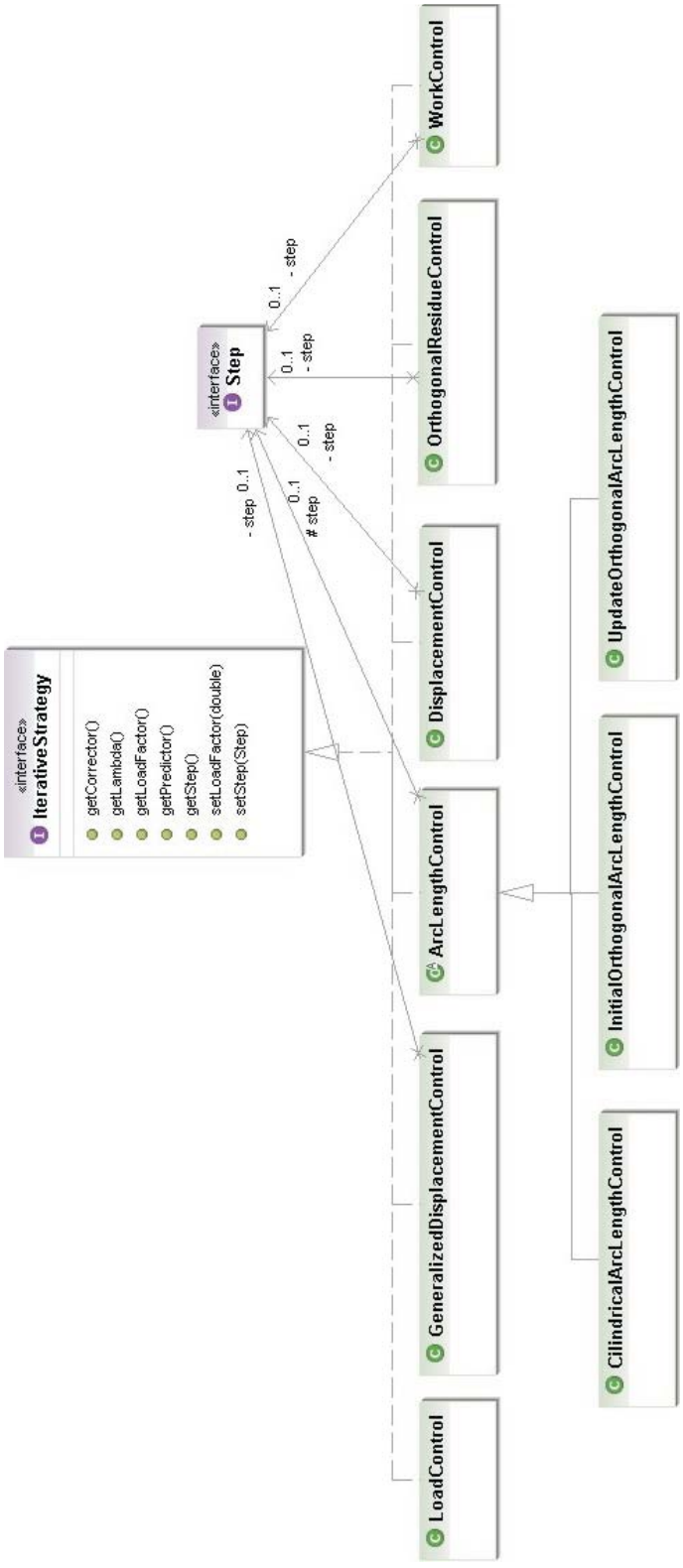

Figure 6: Class diagram for `IterativeStrategy`.

3.3 Model Interface

The discrete model to be analysed is represented in the numerical nucleus by the interface `Model` (Fig. 8). The class which implements `Model`, to represent in the most general way possible a discrete model, is formed by object lists and contains access and manipulation methods for these lists.

`Model` is implemented by the class `FemModel` which represents the finite element model. One object `FemModel` has lists of nodes, elements, shape functions, integration orders, load cases, load combinations, analysis models, materials, constitutive models and degenerations. It has also two attributes: a global analysis model of the type `AnalysisModel` and an object `ProblemDriver`. `FemModel` also extends the class `Observable`, because it is observed by `Persistence`.

Together to the `Model` interface the classes `Node`, `Element` and `ProblemDriver` are implemented. Figure 7 shows the UML diagram for the class `Node`, which has one label and a collection of values of type `java.util.Hash- Map` which stores all the node variables, as, for example, displacements and loads. It extends the class `IPoint3d` and, consequently, has all its methods and attributes, `Node` has also as an attribute one object of this type representing a point in space. The inherited methods are surcharged to access directly the `IPoint3d` object, not the inherited attributes. For example, the method *getCoords()* of `Node` access the object `IPoint3d` and calls its method *getCoords()*, obtaining directly the coordinates of the point in space which represents the node. This process permits the reutilization of code and simplifies the manipulation of an object `Node`.

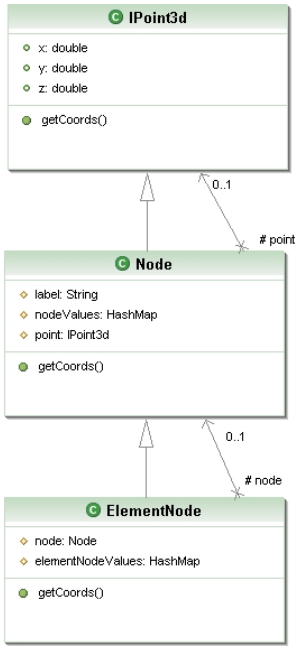

Figure 7: Class diagram for `Node`.

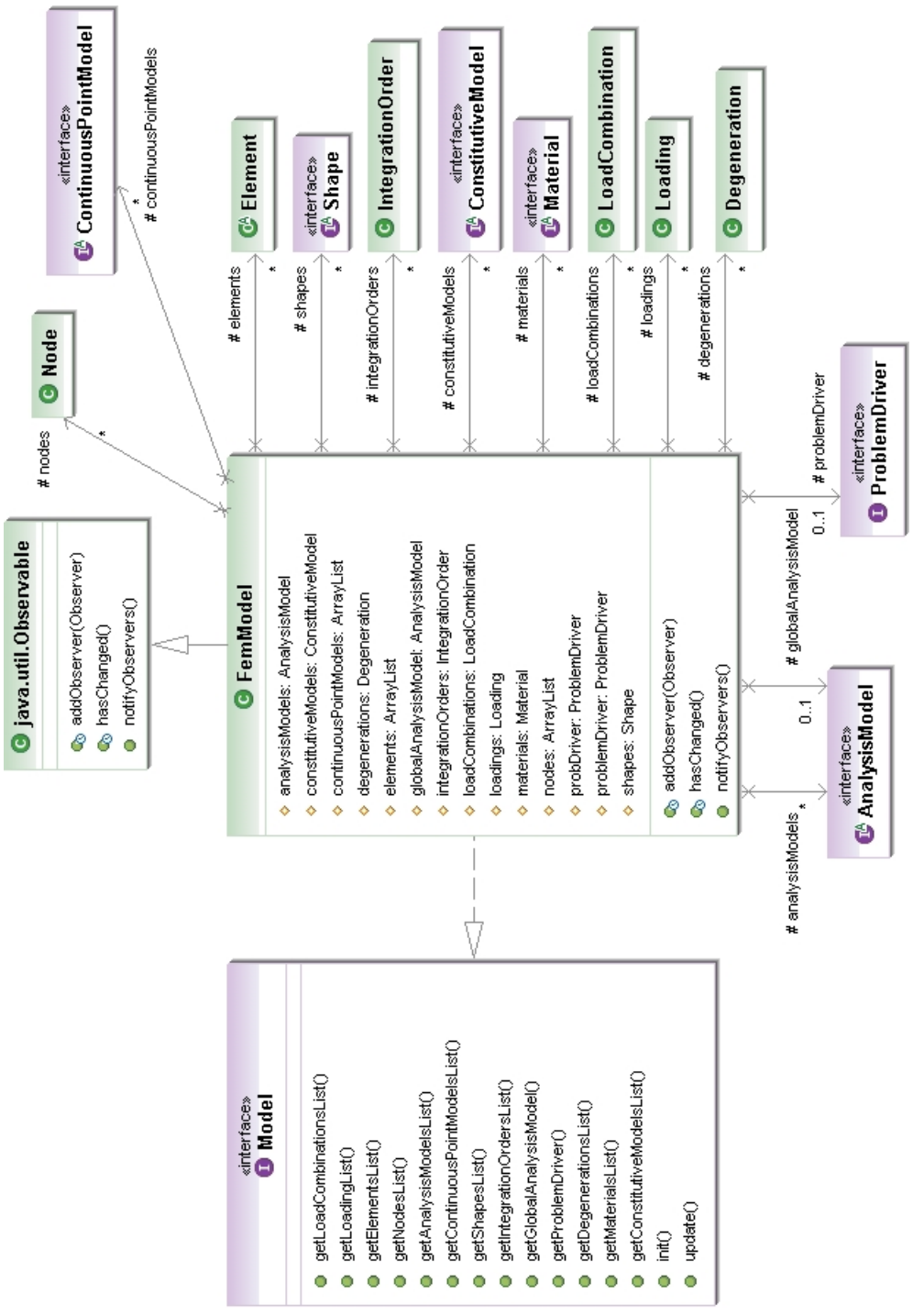

Figure 8: Class diagram for `Model`.

In the same way, as an specialization of node, the class `ElementNode` (Fig. 7) extends `Node` and also has an object `Node`, as well as a collection of values of the type `HashMap`, which stores the equivalent forces to element's body forces. `ElementNode` surcharges the inherited methods, accessing directly the attributes of its object `Node`. For example, its method *getCoords()* calls *getCoords()* of its object `Node`, obtaining the nodal coordinates.

The class `Element` represents the finite elements and is extend by the classes `ParametricElement` and subclasses, `FrameElement` and `ThinPlateElement`, which represent, respectively, the parametric finite elements, framed elements and thin plate elements, as shown in Figure 9. One object `Element` has as attributes a list of `ElementNode`, which represents its incidence, a list of `Degeneration`, which represents its integration points and their geometrical and physical constitution, an object `AnalysisModel`, which represents its analysis model, an object `Shape`, which represents its shape function, an object `ConstitutiveModel`, which represents its constitutive model, and an object `ProblemDriver`, which stores information relative to the type of problem that the element models.

The class `ParametricElement` has, besides `Element`'s attributes, one object `IntegrationOrder` which represents its numerical integration order. In its hierarchy are the classes which represent the parametric finite elements, separated accordingly to their geometry: uni-dimensional elements, triangular and quadrilateral plane elements, and tetrahedrical and hexahedrical elements. These sub-classes implement the methods relative to numerical integration (*addDegenerations(*`Degeneration`*)* and *initDegenerations()*). The class `ThinPlateElement` is extended by `TriangularThinPlate`, which represents the triangular thin plate finite elements.

The `ProblemDriver` interface (Fig. 10) has methods necessary to informing `Assembler` the contributions of each element in model equation. In its hierarchy are represented the various problem types which can be resolved by discrete models. In this way, `Element` is a very general class, independent of the problem which it can represent, but that is capable of passing all necessary information to the model.

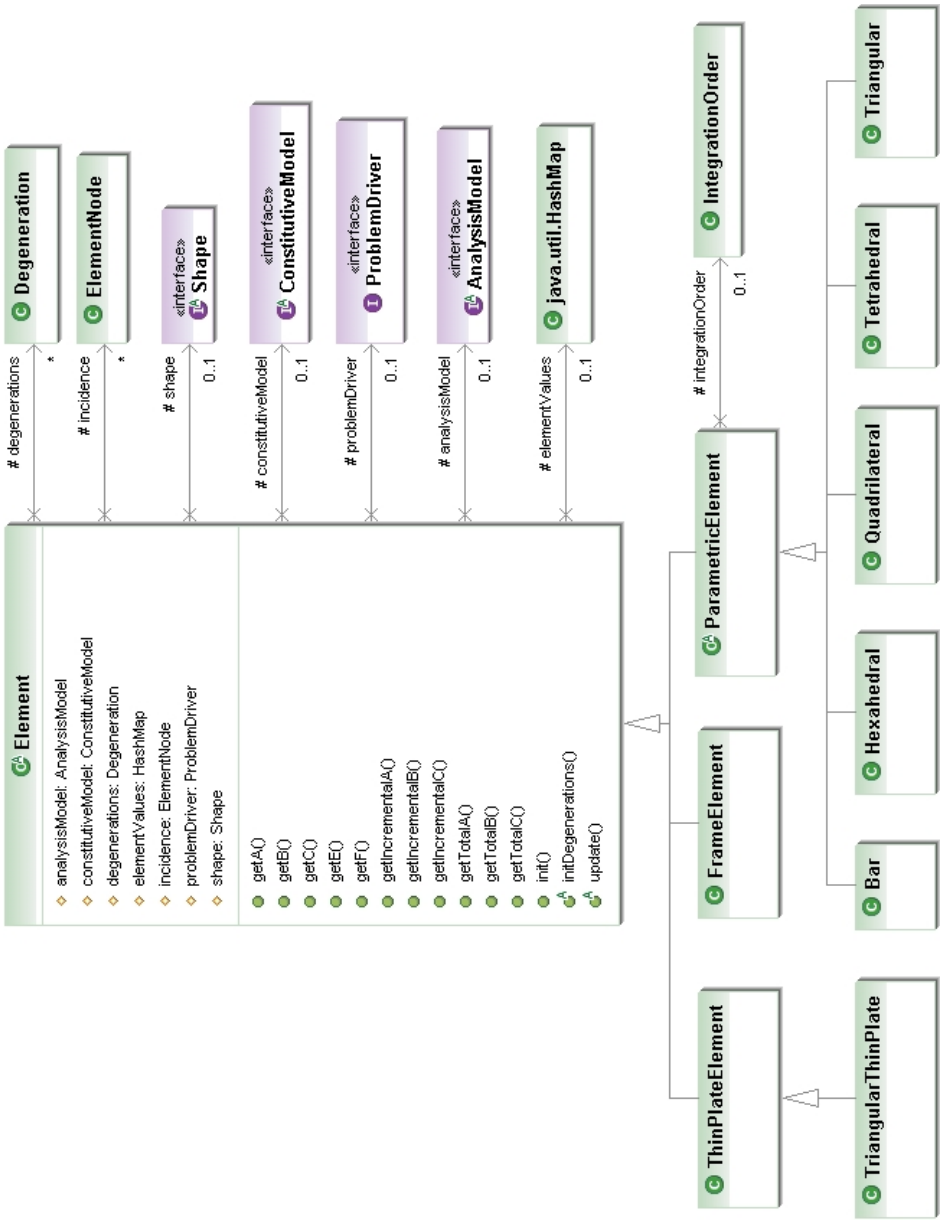

Figure 9: Class diagram for `Element`.

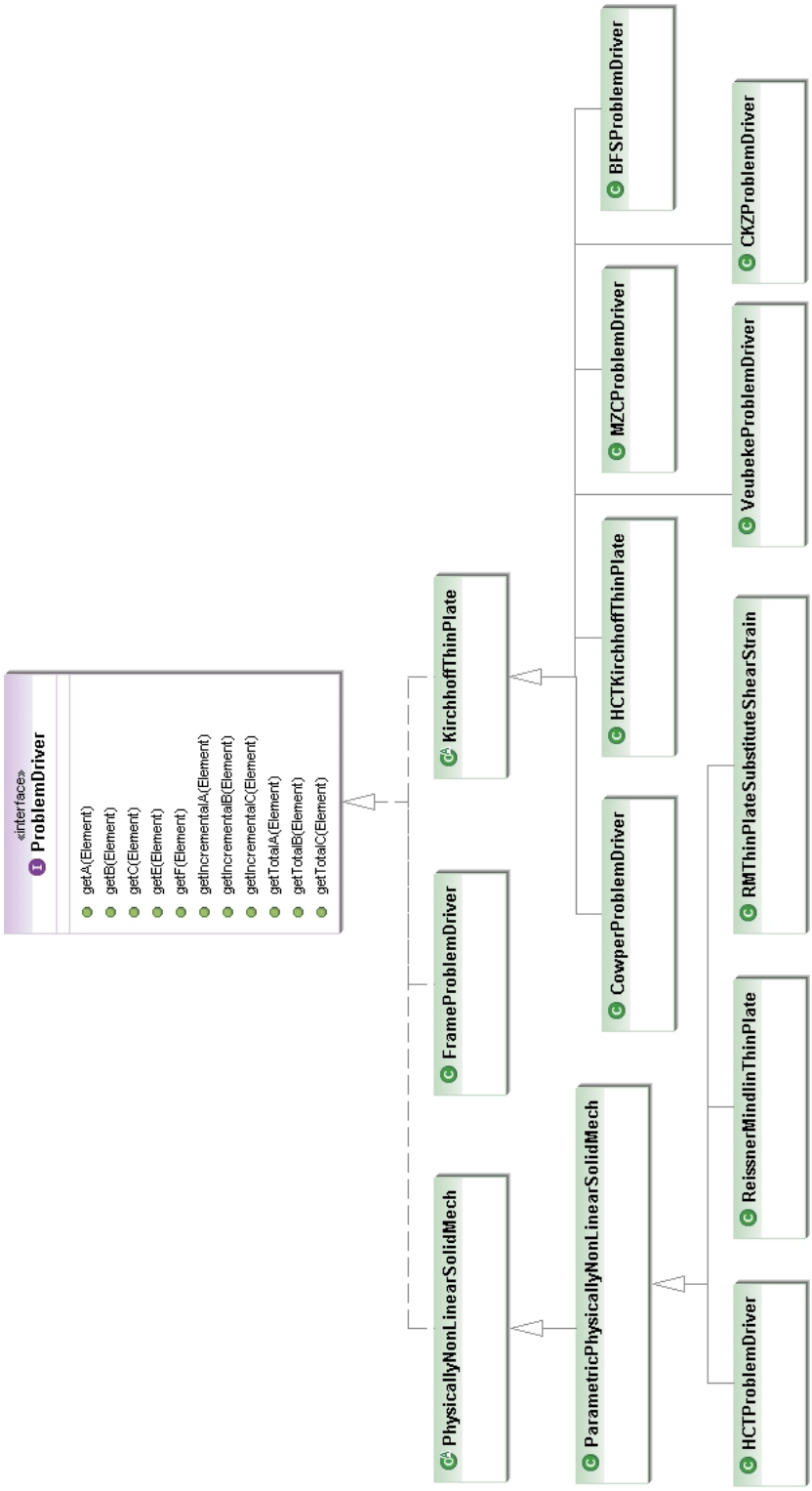

Figure 10: Class diagram for `ProblemDriver`.

4 DYNAMIC GEOMETRICALLY NON-LINEAR ANALYSIS EXPANSION

To implement the dynamic solution, classes which implement the `Solution` interface will be created. For the Newmark-β method, a class named `DynamicIntegrationSolution` will be implemented. In this way, the diagram of Figure 4 will be modified as shown in Figure 12.

It will be also necessary to modify the classes which implement `Assembler` (Fig. 3) in a way to make them capable of mounting the mass and damping matrices. A class named `DynamicLoading`, which inherits the class `Loading`, will be created to represent, as its name says, a dynamic loading, as shown in Figure 11.

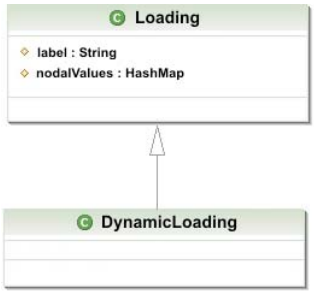

Figure 11: Modified class diagram for `Loading`.

A class implementing the interface `ProblemDriver` and named `GeometricallyNonLinearSolidMech` will also be created. This class will be the responsible for the geometrically non-linear analysis. It will be very similar to the already existent `PhysicallyNonLinearSolidMech` class and it will be able to calculate the mass, damping and stiffness (incremental and total formats) matrices for each element. Figure 13 shows the modified diagram.

The `Step` interface (Fig. 5) will be implemented by a class name `ModifiedNewtonRaphson` which will represent the modified Newton-Raphson incremental-iterative method.

Naturally, the classes that implement the interface `Persistence` will also be modified to include the new attributes and results.

5 CONCLUSIONS

INSANE is a software specially designed to be constantly updated and progressively improved. Many works have been done with this objective, and all of them achieved success.

The expansion of INSANE's numerical nucleus proposed in this paper, aiming to add the capability of performing a dynamic geometrically non-linear analysis of structures, is possible because of the object oriented project conceived at the beginning of the development of this software.

In a few years, INSANE will be a mature, tested and complete structural analysis software, with many analysis models and solutions types available.

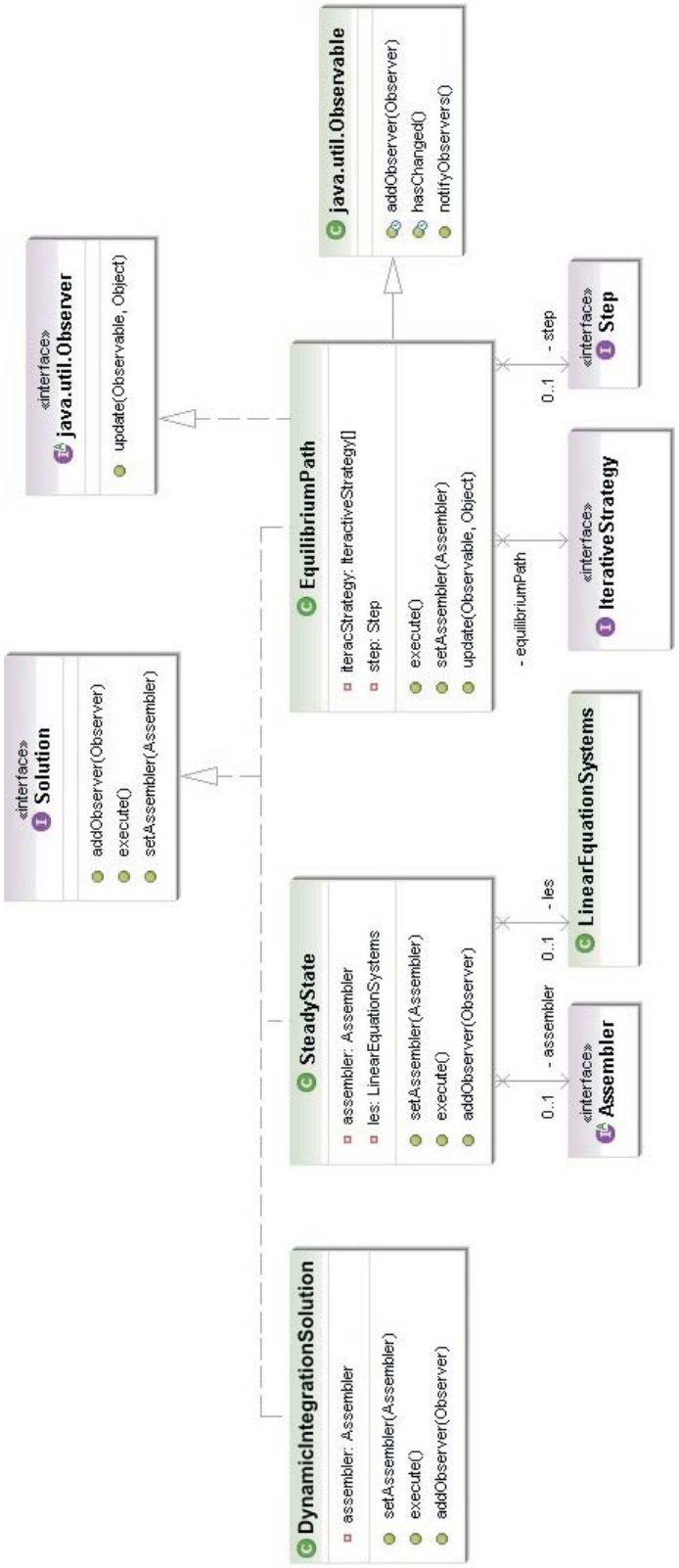

Figure 12: Modified class diagram for `Solution`.

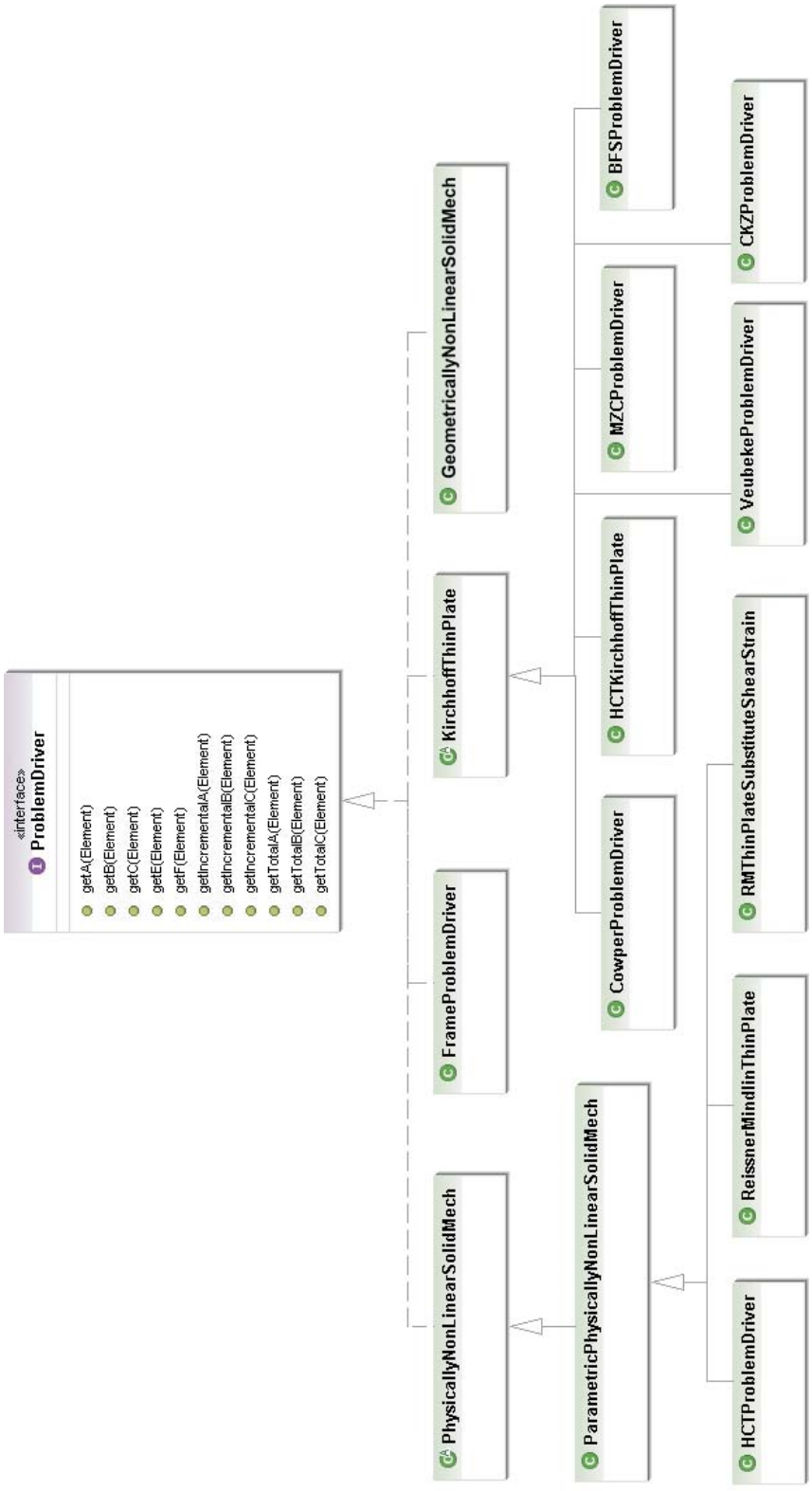

Figure 13: Modified class diagram for `ProblemDriver`.

REFERENCES

[1] W. Weaver Jr. and P.R. Johnston, *Structural dynamics by finite elements*, Prentice-Hall, (1987).

[2] K.J. Bathe, *Finite element procedures in engineering analysis*, Prentice-Hall, (1995).

[3] F.T. Fonseca, R.L.S. Pitangueira and A. Vasconcellos Filho, *Implementação de modelos estruturais de barras como casos particulares do método de elementos finitos*, *Simpósio Mineiro de Mecânica Computacional, Itajubá, Brazil, 2004*, SIMMEC, Itajubá (2004).

[4] J.S. Fuina, *Métodos de controle de deformações para análise não-linear de estruturas*, Master Thesis, UFMG, (2004).

[5] M.L. Almeida, *Elementos finitos paramétricos implementados em Java*, Master Thesis, UFMG, (2005).

[6] L. Germanio, *Implementação orientada a objetos da solução de problemas estruturais dinâmicos via método dos elementos finitos*, Master Thesis, UFMG, (2005).

[7] M.T. Fonseca, *Aplicação orientada a objetos para análise fisicamente não-linear com modelos reticulados de seções transversais compostas*, Master Thesis, UFMG, (2006).

[8] S.S. Saliba, *Implementação computacional e análise crítica de elementos finitos de placas*, Master Thesis, UFMG, (2007).