Programa de Pós-Graduação em Engenharia Elétrica

Escola de Engenharia da Universidade Federal de Minas Gerais

Desenvolvimento de bibliotecas de classes (*frameworks*) para análise de sistemas através do método de Elementos Finitos

Dissertação de mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, como requisito para a obtenção de título de Mestre em Engenharia Elétrica

Autor: Rogério Lopes Braga

Orientador: Professor Renato Cardoso Mesquita

Belo Horizonte, 13 de outubro de 2000

Aos meus pais, Florentino e Edwirgens, por eu ter chegado até aqui.

Ao meu irmão Renato, pelo companheirismo.

No final do jogo, todos,

do Rei ao Peão,

retornam para dentro do mesmo tabuleiro.

Ditado Popular Chinês

Agradecimentos

À Equipe do Programa de Pós-Graduação em Engenharia Elétrica, PPGEE, pela sua incontestável compreensão e auxílio nas dificuldades.

Ao Prof. Renato Cardoso Mesquita pela sua indiscutível capacidade técnica, competência e orientação.

Ao Prof. Rodney Rezende Saldanha pelas inúmeras vezes que auxiliou no andamento e na finalização dos testes.

Aos amigos do CPDEE, em especial, Márcio Matias, Antônia Navarro e Ana Liddy, pelas longas horas de estudo e dedicação.

Aos meus tios, José Lage e Rosalba, pelo apoio durante o curso, Roberto e Tina, pela ajuda na finalização dos trabalhos.

Sumário

Resumo

Esse trabalho contempla o desenvolvimento de um sistema, totalmente orientado a objetos, que calcula, utilizando o método de Elementos Finitos, propriedades eletromagnéticas em uma superfície definida.

O cálculo pelo método de Elementos Finitos tem como característica importante sempre manter os mesmos passos básicos para qualquer que seja o tipo de problema que está sendo resolvido. Essa característica, somada a tecnologia que está sendo utilizada, dá ao sistema a possibilidade de chegar a ser um framework, definindo, então, o objetivo do nosso trabalho: o desenvolvimento de um sistema, seguindo diretrizes que o levem a ser um framework.

Seguindo os princípios gerais para a criação de um framework, estudamos as aplicações existentes e extraímos dessas a estrutura básica do nosso sistema, ou seja, a parte comum existente entre essas aplicações.

Depois de extraída essa estrutura, foram estudadas e utilizadas técnicas de implementação, proporcionadas pelo desenvolvimento orientado a objetos, como a herança, o polimorfismo e o despacho de funções. Essas técnicas proporcionaram a implementação da estrutura genérica conseguida através do estudo das aplicações.

A generalidade dessa implementação, proporcionada pela estrutura e pelas características do método, gerou classes de fácil entendimento e manipulação.

Depois da estrutura básica implementada, o trabalho foi dividido em etapas de desenvolvimento onde, cada etapa gerava um novo protótipo da aplicação, agregando mais informações à estrutura existente. Durante o desenvolvimento dessas etapas foram mostradas vantagens advindas dos frameworks: facilidade de implementação de novas características e abundância de testes, já que o mesmo fluxo básico é sempre executado para todos os tipos de problemas.

Finalizando, geramos um produto muito bem documentado, de fácil entendimento, pela generalidade, e com todas as demais características obtidas pelo fato dele ser um framework.

Abstract

This work shows an object oriented system development. This system implements the Finite Elements Method to calculate electromagnetics features on a defined surface.

The Finite Elements method has a very important feature: this method always performs the same basic steps to solve all kinds of problems. This feature, added to the technology that has been used, gives the system the possibility to become a framework. That is this the work goal: the development of a framework for the finite elements method.

Following the general bases to build a framework, we have studied some existent application and extract the basic structure of the system from them.

Various object oriented techniques were used to implement the framework: the inheritance, the polymorphism and the functions dispatch.

Classes of easily understanding and manipulation were developed, generating a very general implementation.

The development cicle was splited into parts, with part consisting of a new application prototype, adding more information to the existent structure. During the development, of those parts, the advantages of frameworks clearly appeared: easily implementation of new features and a variety of tests, because the same basic structure is always executed to all kinds of problems.

The final product is very well documented, is easily to be understanding, because of generality, and with all the other features gotten by the frameworks.

1 - O que é um framework

Existe um número bastante vasto de conceitos similares para definir frameworks:

a. “Um framework envolve certas escolhas sobre estados de particionamento e controle de fluxo; a reutilização completa o framework para produzir uma aplicação” [Deu89].

A definição de Johnson e Foote [Jon98] é uma das mais conhecidas:

b. ”Um framework é um conjunto de classes que envolve um desenho abstrato para solucionar uma família de problemas relativos a um determinado domínio.”

Uma definição parecida foi apresentada por Johnson anteriormente [Joh91]:

c. “Um framework é um conjunto de objetos que colaboram e conduzem um conjunto de responsabilidades de uma aplicação ou um domínio de um subsistema.”

Firesmith dá a seguinte definição [Fir94]:

d. “Um framework é uma coleção de classes colaborativas que encapsulam as estruturas e os principais mecanismos comuns requeridos e projetados para um domínio específico de uma aplicação.”

Baseando-nos nas definições encontradas na literatura, sugerimos a seguinte:

Framework é um conjunto de classes interligadas, que possuem seu próprio fluxo de controle de desenvolvimento. O conjunto de classes e o fluxo de controle devem ser bastante genéricos, proporcionando ampla reutilização na evolução do sistema. Um dos maiores objetivos dos frameworks é possibilitar a reutilização de código, facilitando a implementação de novas aplicações que utilizem o domínio para o qual o framework foi desenvolvido. A reutilização leva à facilidade de evolução, testes e manutenção do framework.

1.1 Motivações para o presente trabalho

O método de Elementos Finitos desenvolve uma seqüência de passos que nunca é alterada. Escreveremos um pouco sobre essa seqüência, com o objetivo de mostrar que o método se adapta aos princípios básicos para a criação de frameworks, sendo este um grande fator motivador para o nosso trabalho.

No método de Elementos Finitos, inicialmente os dados que serão trabalhados são lidos de um dispositivo externo de armazenamento, geralmente um arquivo normalizado, e alocados dinamicamente na memória do computador. Esses dados descrevem o problema que será resolvido. No caso específico desse método, o problema é limitado a uma superfície fechada. Essa superfície fechada está dividida em pequenas superfícies chamadas elementos. Os elementos têm formas variáveis, triangulares, quadrangulares, etc, constituindo a base de trabalho do método. A figura 1.1.1 ilustra a superfície de trabalho subdividida em elementos triangulares. Nesse caso, o método está sendo utilizado para calcular campos magnéticos em um motor. O objetivo dessa figura é ilustrar a base do método, que são os elementos com seus respectivos nós, neste caso, os vértices dos triângulos.

Uma das características principais dos nós são as suas coordenadas, grandezas que são necessárias para os cálculos. Nesse trabalho implementaremos também os elementos de segunda ordem ou de ordem superior. A diferença desses elementos para os elementos mostrados nas figuras é simplesmente a de possuírem mais nós e trabalharem com funções de interpolação de ordem mais elevada, aumentando a precisão dos cálculos. Nessa superfície existem também as fontes, que no nosso caso específico podem ser elétricas ou magnéticas. O método de Elementos Finitos é utilizado para calcular a distribuição dos campos no domínio. Para a estimação desses valores são calculadas a contribuição de cada fonte nos elementos, e então essas contribuições são condensadas em uma matriz e em um vetor globais. A solução do sistema matricial resultante fornece os potenciais nos diversos nós [Jim93].

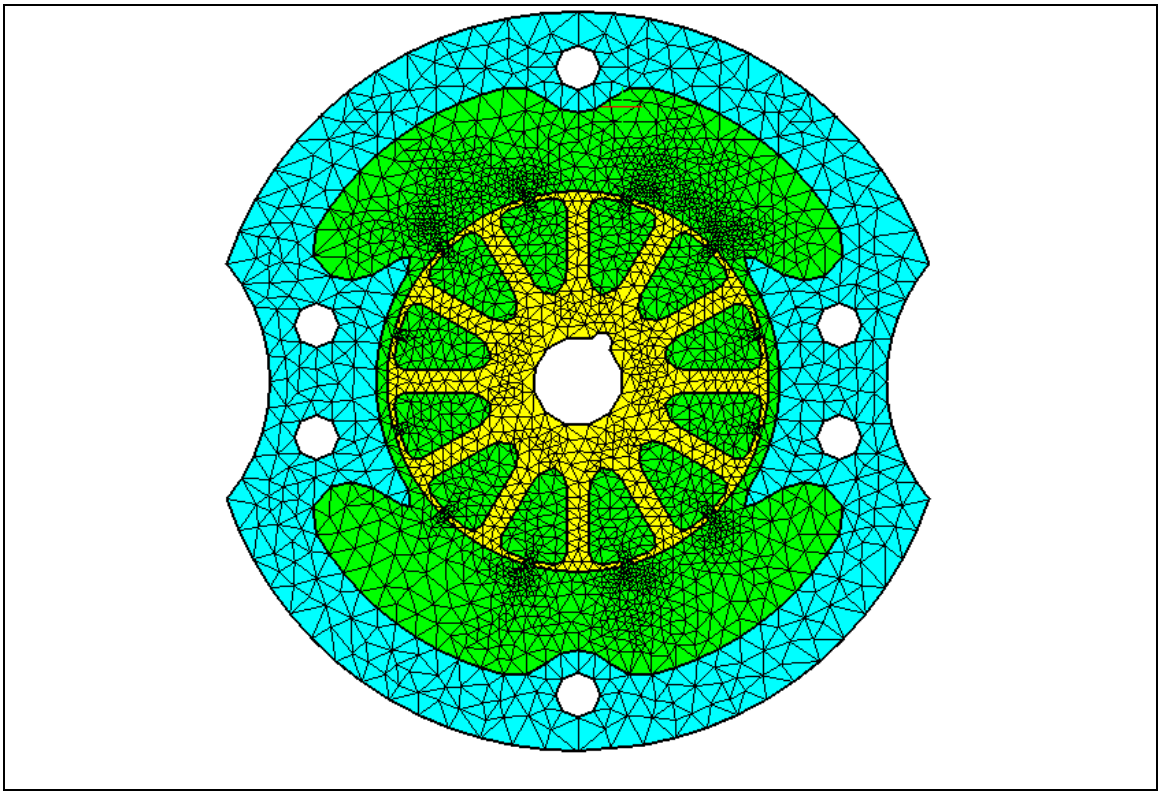

Figura 1.1.1: Malhas de elementos finitos em um motor elétrico

Pode-se verificar que o método de Elementos Finitos executa sempre a mesma seqüência de passos, independentemente da forma dos elementos, das dimensões do problema 1D, 2D ou 3D e do tipo de problema. Ele simplesmente lê um arquivo de entrada, armazena seus dados em memória, calcula as contribuições de cada elemento, agrupa essas contribuições em uma matriz e em um vetor globais e soluciona o sistema matricial resultante [Jim93].

Estudando essa seqüência, verificamos que ela se adapta perfeitamente às características de um framework. Independentemente do tipo de problema, o método possui o mesmo fluxo de desenvolvimento. Os problemas possuem diferenças apenas nas suas formulações e tipos de elementos que utilizam. Existem muitas funções comuns, possibilitando a geração de funções genéricas. As especificidades são resolvidas perfeitamente com o polimorfismo, capacidade que permite serem definidas, em tempo de execução, algumas características para solução de problemas [Koe97].

Generalizando totalmente os problemas que utilizam o método, o que conseguimos visualizar é a estrutura de um framework. Dessa forma, estima-se que será obtido um produto eficiente, de alta facilidade para manutenção e evolução.

Com a geração do framework em C++, pretendemos ainda obter possibilidade de transferência entre plataformas, incluindo as de alto desempenho. Finalizando, esperamos que esse framework tenha grande utilidade para futuros projetos que envolvam Elementos Finitos.

1.1.1 A necessidade dos frameworks

Os problemas com a reutilização de classes que não possuem as características dos frameworks são os seguintes [Mat96]:

a. Complexidade: Em um grande sistema a hierarquia das classes pode ser muito confusa. Se o analista não tem acesso a uma documentação detalhada, que é princípio básico dos frameworks, o aproveitamento do código já escrito será muito difícil.

b. Fluxo de controle: O controle do fluxo do programa tem que ser administrado pelo analista, o que também não acontece com o framework, já que o fluxo faz parte de uma classe que tem por fim esse controle.

c. Duplicação: O uso de grupos de classes, permite aos programadores reutilizar classes de maneiras diferentes.

Com o emprego da teoria para geração dos frameworks, teremos:

a. Uma grande redução no número de linhas e funções a serem implementadas. As aplicações herdam grande parte já implementada nos frameworks.

b. Tudo que se herda de um framework possui um fluxo e já foi testado.

c. O framework oferece reutilização de estrutura e não somente reutilização de código. Existe uma transferência de conhecimento quando se utiliza um framework.

d. Aspectos eficientes do software também são herdados automaticamente com a utilização dos frameworks.

e. Ampliação da facilidade de manutenção: quando um erro é corrigido em um framework, a mesma correção é herdada por todas as aplicações que utilizam aquele framework [Cot95].

É interessante citar também que os frameworks possuem desvantagens:

a. É mais difícil projetar um bom framework. É necessário um grande conhecimento do universo que se está trabalhando para que se possa implementar todas as suas características.

b. A documentação é crucial para a reutilização dos frameworks, e, a boa documentação é difícil de ser mantida.

c. Toda a evolução dos frameworks é herdada pelas suas aplicações. Muitas vezes o efeito dessa herança pode não ser o esperado, quebrando a compatibilidade.

d. A generalidade pode prejudicar a eficiência. Quando generalizamos estamos criando novas estruturas. Essas estruturas, muitas vezes, significam mais acessos a periféricos, mais alocação de memória, mais gerenciamentos, maior indireção na chamada de funções, etc.

e. É mais difícil distinguir os problemas. Eles podem estar tanto nas aplicações quanto no próprio framework. Muitas vezes o analista das aplicações não tem acesso ao código do framework.

1.1.2 Situação Atual

Para iniciarmos o desenvolvimento desse framework fizemos uma vasta pesquisa. Encontramos softwares que utilizam o método de Elementos Finitos, mas não encontramos um framework que o implementa. Os códigos encontrados muitas vezes são híbridos, utilizam C++, C e ainda rotinas em Fortran, não tendo o objetivo de implementar um framework.

A literatura verificada mostra que existe uma grande tendência à utilização dos frameworks [Mat96] [Deu89] [Fir94]. As facilidades decorrentes do seu uso fortalecem a sua ramificação por diversas áreas científicas. Quanto mais

complexas as aplicações, maior a necessidade de encapsulamento e aproveitamento de código já testado. Essas facilidades proporcionadas pelos frameworks facilitam muito o desenvolvimento de sistemas .

1.2 Objetivos do trabalho

Esse histórico reforça a viabilidade do nosso objetivo: vamos gerar um framework, que utiliza o método de Elementos Finitos, com toda a estrutura básica para solucionar problemas de engenharia. Este framework deve ter as seguintes características: deve ser genérico o suficiente para poder contemplar o maior número possível de problemas resolvidos por Elementos Finitos, deve proporcionar ampla facilidade na inclusão de novas características do método e deve estar bem documentado, proporcionando ampla facilidade no seu entendimento. Atendendo a esses requisitos, o produto conseguido será uma grande contribuição para a comunidade científica da área.

1.3 Organização da dissertação

Este trabalho está subdividido em quatro capítulos.

Capítulo 1: são feitas considerações gerais e definidos conceitos básicos necessários à compreensão e desenvolvimento do trabalho.

Capítulo 2: faz algumas considerações sobre teorias e técnicas de modelagem de frameworks. É dada ênfase às técnicas e softwares de documentação que são utilizados neste trabalho.

Capítulo 3: apresenta todo o processo de implementação que foi mostrado teoricamente no capítulo 2, desenvolvendo o framework passo a passo.

Capítulo 4: conclusão - faz algumas considerações sobre o desenvolvimento do trabalho, mostra as suas contribuições e as possibilidades de trabalhos futuros.

2 - Teorias e técnicas para implementação de frameworks

2.1 Considerações iniciais

Seguindo o nosso roteiro, estudaremos nesse capítulo algumas técnicas que auxiliam o desenvolvimento de frameworks. Mostraremos duas características importantes do desenvolvimento: uma é a experiência no problema que se quer resolver, pois quanto maior o conhecimento, maior a capacidade de generalização e visão das limitações para implementação do framework; a outra é a experiência com desenvolvimento de frameworks, pois as pessoas, que já estão acostumadas com o desenvolvimento, selecionam caminhos que trazem menores problemas para a implementação.

2.2 Desenvolvimento

Existem vários processos para o desenvolvimento dos frameworks [Wil93] [Joh93]:

a. Desenvolvimento baseado na experiência adquirida com as aplicações que se quer desenvolver.

Primeiro desenvolvemos um número *n* de aplicações dentro do contexto do problema. Quando as aplicações estiverem funcionando corretamente, começamos com a primeira interação. Devemos identificar as características comuns das aplicações e colocá-las no framework. Para verificar se as características estão ou não corretas, devemos refazer as aplicações utilizando o framework. Esta tarefa será bem fácil se as características estiverem corretas. Se houver problemas, devemos identificar novamente as características comuns das aplicações e utilizar a experiência adquirida até desenvolvermos uma segunda versão do framework. Baseadas na nova versão

do framework, novas aplicações podem ser desenvolvidas. A experiência que vem sendo adquirida com o desenvolvimento de novas aplicações são usadas como entrada para a manutenção do framework. Esse tipo de desenvolvimento é exemplificado pela figura 2.2.1:

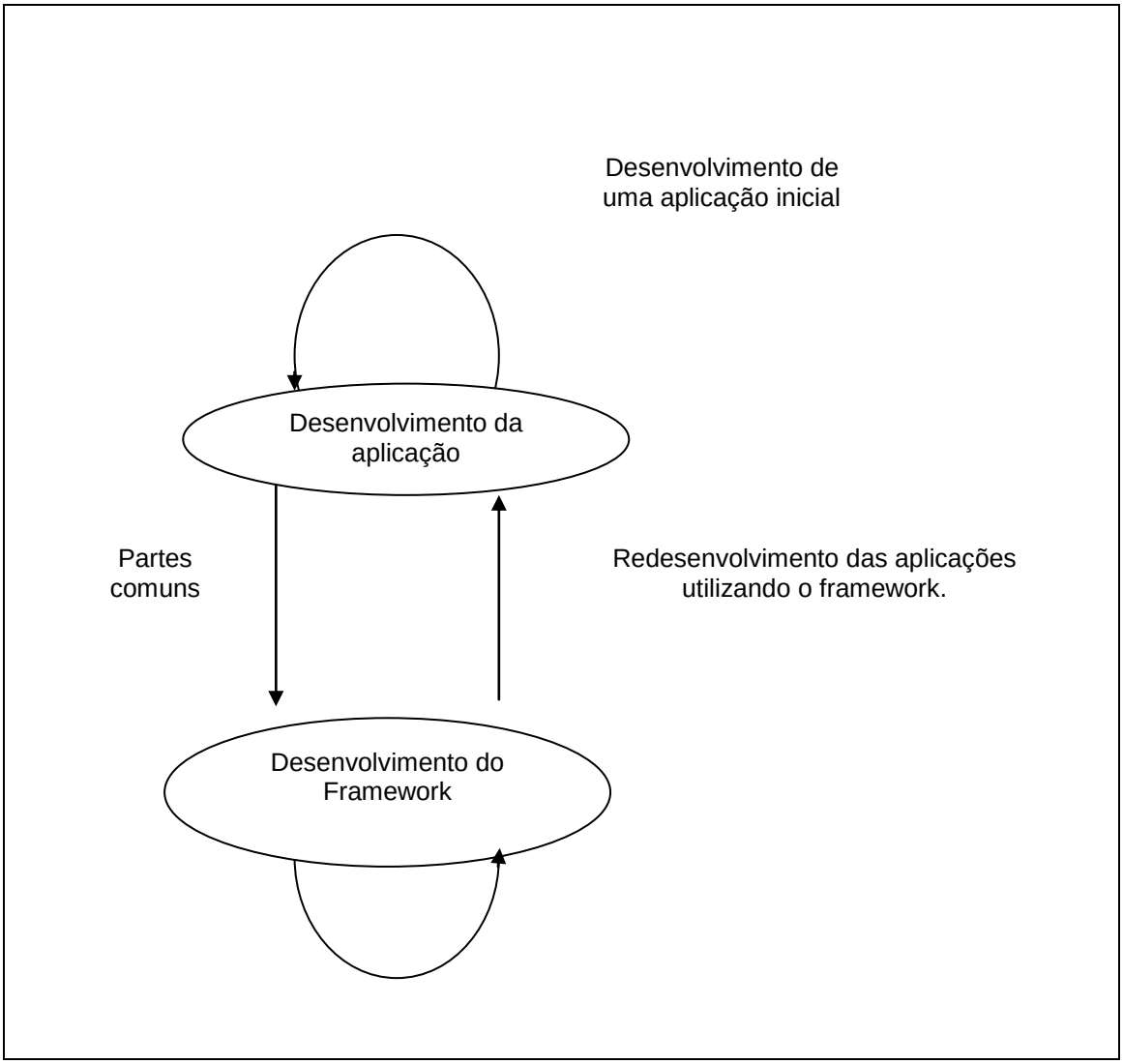

Figura 2.2.1: Desenvolvimento baseado na experiência adquirida com as aplicações que se quer desenvolver

b. Desenvolvimento baseado na análise do domínio que o framework terá.

A primeira atividade é analisar o domínio do problema para identificar, entender e conseguir um nível adequado de abstração. Analisar o domínio significa analisar as aplicações existentes e isto só é possível se existirem aplicações desenvolvidas para o problema. A análise das aplicações existentes

mostrará uma grande parte do código que será utilizado pelo framework. Depois de terem sido conseguidas as abstrações e terem sido identificadas as partes que são comuns, começamos o desenvolvimento do framework junto com uma aplicação de teste. A seguir, desenvolvemos uma segunda aplicação baseada no framework. Devemos verificar se o framework está funcionando para as duas aplicações. Se necessário, deve ser feita uma revisão no framework para atender a todas as aplicações. Esse tipo de desenvolvimento é exemplificado pela figura 2.2.2:

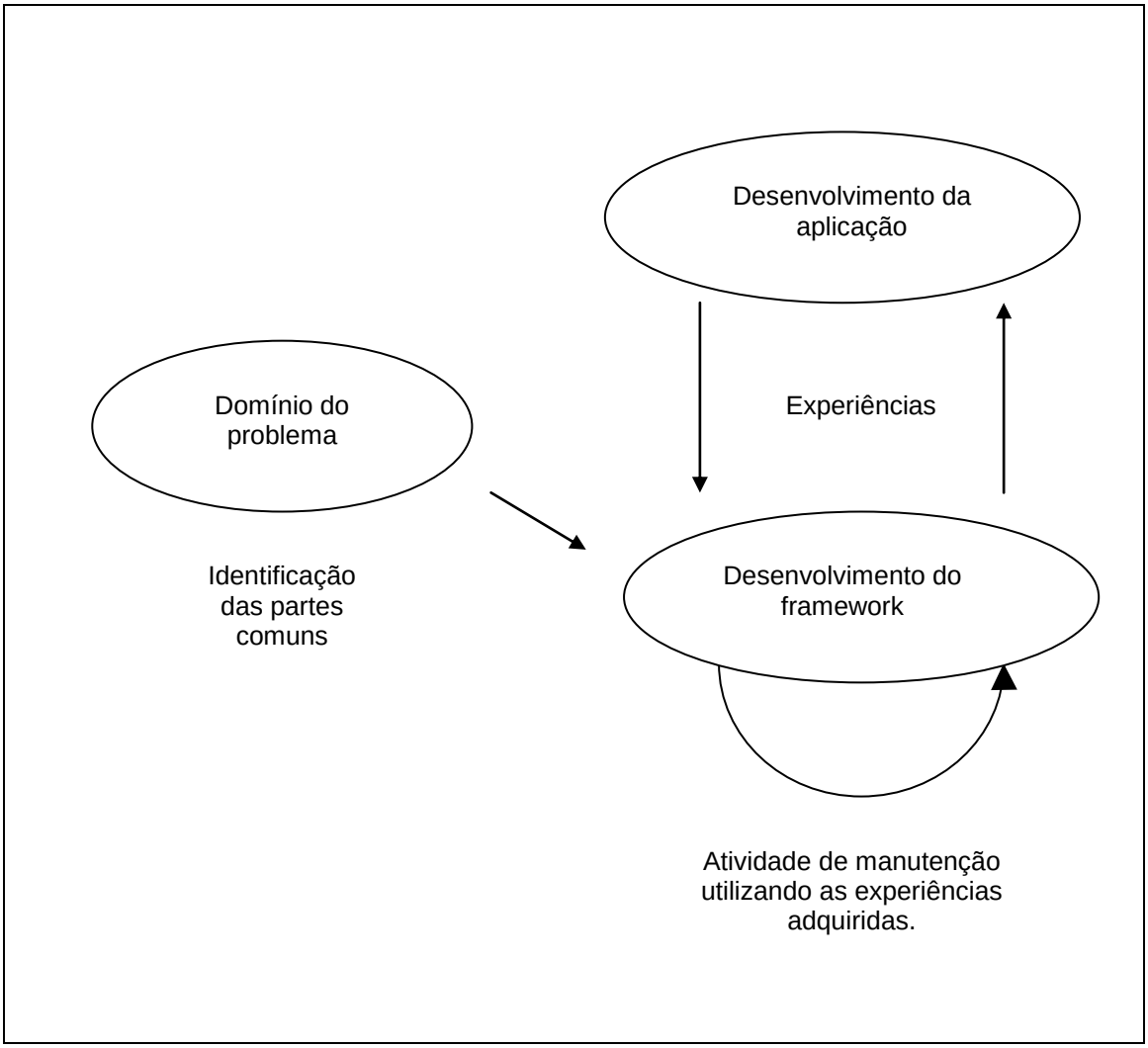

Figura 2.2.2: Desenvolvimento baseado na análise do domínio que o framework terá

c. Desenvolvimento utilizando as raízes da estrutura.

Primeiro desenvolvemos uma aplicação que contenha o domínio do problema. Segundo, estabelecemos a base de apoio necessária para gerar as raízes da estrutura. Utilizamos a base conseguida e então criamos o framework. A interação entre aplicação e framework começa a acontecer. Neste processo também são agendadas atividades de manutenção no framework. Esse tipo de desenvolvimento é mostrado na figura 2.2.3:

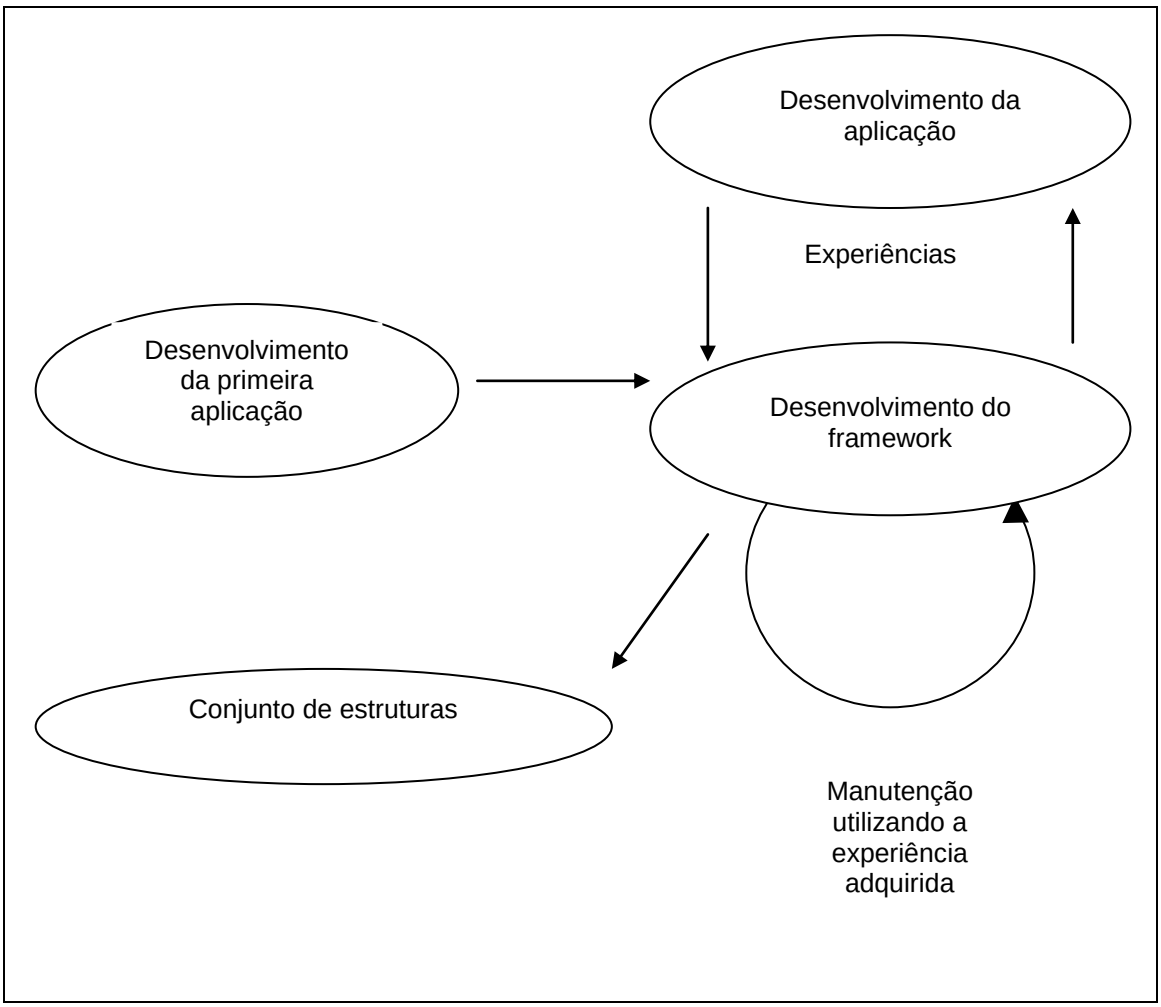

Figura 2.2.3: Desenvolvimento utilizando as raízes da estrutura

Generalizando os elementos comuns dos processos mostrados temos:

a. Análise do domínio do problema. Isto é conseguido sistematicamente com o desenvolvimento de algumas aplicações, utilizando o domínio para que as abstrações sejam alcançadas.

b. A primeira versão do framework é obtida, utilizando as abstrações conseguidas.

c. Desenvolvemos, então, mais aplicações utilizando o framework. Isto é, a atividade de teste do framework. Com as novas aplicações é que se vê se o framework está sendo realmente aproveitado.

d. Os problemas no framework, que vão aparecendo durante o desenvolvimento de novas aplicações, são resolvidos durante o desenvolvimento das próximas versões.

e. Depois desse ciclo ter sido repetido algumas vezes, podemos dizer que o framework está adquirindo maturidade.

Essa generalização dos elementos comuns é mostrada na figura 2.2.4:

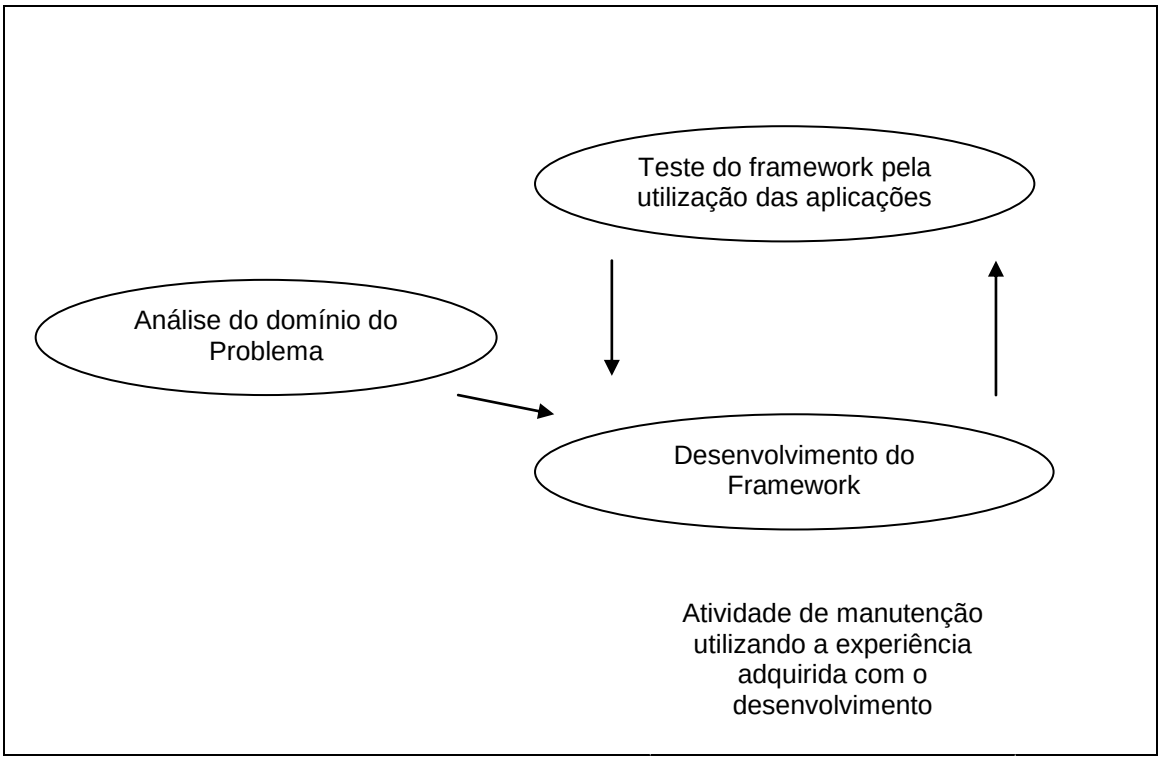

Figura 2.2.4: Representação genérica do desenvolvimento de um framework

2.3 Princípios para o desenvolvimento

Existem algumas referências que mostram princípios para o desenvolvimento de frameworks [Wil93] [Wei89].

Um princípio é a iteração do framework com o usuário do framework. Por exemplo, quando estamos desenvolvendo um framework é necessário determinar limites: o que será tarefa do framework e o que será tarefa dos futuros usuários do framework.

No caso deste trabalho esse princípio pode ser expresso com o seguinte parágrafo:

O framework que está sendo desenvolvido fará a leitura de um arquivo de entrada com os dados que descrevem o problema em formato neutro [Mag00], solucionará o problema, utilizando o método de Elementos Finitos e gravará os resultados em um arquivo de saída. Não terá nenhuma implementação gráfica, ficando todo esse trabalho para os futuros usuários que estarão gerando uma aplicação para o framework.

Outro princípio é modelar o framework independentemente das interfaces e das plataformas: o framework sempre deve ser muito genérico. Para atender esse princípio, a linguagem de codificação deve ser uma linguagem existente na maioria das plataformas, possibilitando assim o transporte entre essas plataformas.

Para atender a esse princípio, a plataforma de desenvolvimento utilizada será a plataforma mais genérica que contenha todas as características da orientação a objetos. Isso possibilitará a transferência, tanto para outros sistemas operacionais quanto para outros ambientes. Utilizamos um compilador C++ padrão, o Borland C++, logicamente, sem utilizar nenhum recurso além dos existentes na padronização ANSI do C++ [Str98] .

A documentação do framework deve ser a mais detalhada possível. Existem softwares para documentação que já possuem todas as características da orientação a objetos, facilitando o trabalho.

Para a documentação do projeto do framework adotamos o Rational Rose [Qua98], software que vem sendo utilizado pelo Grupo de Otimização e Projetos Assistidos por Computador - GOPAC - com bons resultados.

2.4 Fases do desenvolvimento

Na figura 2.4.1 são mostradas as fases básicas para o desenvolvimento de um sistema. Essas fases devem ser seguidas e, quando finalizadas, devem ser repetidas. Esta é a base do processo de desenvolvimento de software em especial [Fow97]. As repetições geram as novas versões e, com as novas versões, alcançamos o amadurecimento do sistema.

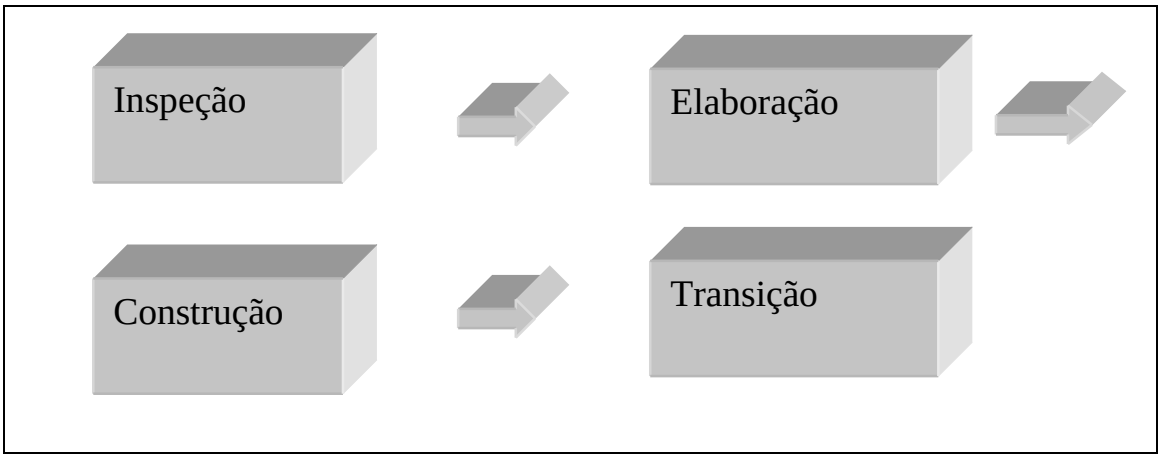

Figura 2.4.1: Fases do desenvolvimento de um sistema

Resumidamente, essas fases são compostas pelas seguintes atividades:

2.4.1 Inspeção

Nesta etapa do desenvolvimento são levantadas todas as informações vitais para o projeto: custo, retorno, etc. Esta fase tenta Diagnosticar a dimensão do problema, verificando então sua viabilidade.

Toda a nossa parte introdutória apresenta o levantamento dessas informações, certificando a viabilidade do trabalho que está sendo desenvolvido.

2.4.2 Elaboração

Essa fase pressupõe um bom conhecimento do problema: não existem dúvidas sobre o que vai ser feito, como será feito e que tecnologia será utilizada. No nosso caso específico, estamos desenvolvendo um framework, biblioteca de classes hierarquizadas e com fluxo próprio, que solucionará problemas de eletromagnetismo dos tipos estáticos e quase estáticos. Os problemas podem ser resolvidos para uma e duas dimensões. Usaremos para a solução desses problemas o já citado Método dos Elementos Finitos [Hug87] [Jim93]. Quanto a tecnologia, como foi dito durante a explicação dos princípios para a modelagem de frameworks, adotamos o compilador Borland C++ utilizando somente os recursos existentes na padronização ANSI do C++.

O próximo passo dentro da elaboração é tentar diagnosticar as estruturas e funções do sistema, pelo menos as principais, como por exemplo: as de leitura do arquivo de entrada, de alocação de memória dinâmica, as árvores genéricas de armazenamento de dados e suas funções de manipulação e as funções para execução dos passos necessários para a solução do sistema. Este detalhamento do sistema pode ser feito com a utilização de papel e lápis, mas ele requer constantes mudanças e essas mudanças têm que ser implementadas de maneira rápida, tornando o papel e lápis não adequados.

Por esse motivo começaram a surgir técnicas que criam um certo padrão e auxiliam no desenvolvimento dos sistemas. A UML, The Unified Modeling Language [Fow97], por exemplo, é uma linguagem industrial para especificação, visualização, construção e documentação de sistemas. Utilizando a UML, programadores e analistas têm uma padronização no trabalho, tornando o desenvolvimento do sistema mais simples. A UML foi criada pelos metodologistas Grady Booch, Ivar Jacobson e Jim Rumbaugh e é padrão atual para a modelagem orientada a objetos e, por isto, foi adotada neste trabalho.

Uma das técnicas presentes na UML são os Casos de Uso [Fow97]. Os Casos de Uso podem ser entendidos como os vários cenários necessários para descrever as seqüências que são executadas em um sistema. Nesse ponto, os casos de uso ainda não precisam ser amplamente detalhados. Além dos casos

de uso, a UML [Qua98] proporciona a possibilidade de detalhar ainda mais o sistema através de diagramas como: diagramas de Classes, diagramas de Seqüência, diagramas de Colaboração, diagramas de Estados, etc. É importante lembrar que cada diagrama proporciona uma visão específica do sistema. Dependendo do sistema, é necessário o desenvolvimento de todos os diagramas para o seu completo entendimento. Para outros sistemas, nem todos os diagramas são necessários: tudo depende das visões que são importantes para a compreensão do sistema.

A fase de elaboração está finalizada quando pudermos precisar o tempo para implementar todos os casos de uso e quando todos os problemas identificados são passíveis de serem resolvidos.

2.4.3 Construção

Dividimos o sistema em uma série de minissistemas. Para cada um dos minissistemas executamos as seguintes rotinas: Análise, Desenho, Codificação e Testes, com a respectiva comparação com os casos de uso. Quando todos os minissistemas estiverem testados, o mesmo processo deve ser executado para todo o sistema, fechando esse ciclo.

2.4.4 Transição

No final de todo o desenvolvimento do projeto, sempre existem alguns itens que não poderiam ser tratados nas fases anteriores como, por exemplo, a otimização. A otimização reduz o entendimento do programa com o objetivo de melhorar a performance. Tratar a otimização nas fases anteriores pode trazer outros tipos de preocupações, atrasando o projeto. É importante frisar que durante essa fase, não devemos adicionar novas funções, a menos que seja essencial. Um bom exemplo de transição é o tempo entre a versão beta e a versão final do sistema.

2.5 Conclusão

Essas fases na realidade são a subdivisão do sistema em níveis onde cada nível possui as suas etapas e a realização de uma depende da boa

finalização da anterior. Seria muito difícil tentar resolver tudo de uma só vez, daí então a necessidade da divisão.

O próximo capítulo mostrará as divisões que foram necessárias para a implementação do trabalho e também a interação entre elas.

3 - Implementação

3.1 Considerações iniciais

Utilizando as técnicas apresentadas no capítulo anterior, mostraremos o desenvolvimento do framework. É importante lembrar que o Grupo de Otimização e Projetos Assistidos por Computador – GOPAC - já possui algumas estruturas e softwares prontos. Esses softwares e as estruturas foram analisados com a intenção de, se possível, ganharmos tempo e também conhecimento. As técnicas mostradas no capítulo anterior foram seguidas. Acrescentamos análise das possibilidades de implementação do método em C++ e também análise da estrutura existente no GOPAC. A análise das possibilidades de implementação tem o objetivo de facilitar o entendimento do trabalho. Já a análise da estrutura existente teve o objetivo de agilizar o trabalho.

Depois de feitas as primeiras análises do projeto, dividimos a fase de construção do nosso framework em etapas. Cada uma dessas etapas gerou um minissistema que foi comparado com as especificações necessárias para a construção de um bom framework, explicitadas no capítulo anterior.

3.2 Analisando o método de Elementos Finitos, avaliando as possibilidades de implementação

É interessante mostrar, nesse ponto do trabalho, como se estruturou o método de Elementos Finitos para que, ao final do desenvolvimento, um framework seja obtido. Como foi descrito no primeiro capítulo, os frameworks possuem o controle dos seus passos, ou seja, o seu próprio fluxo. Por isto, uma classe específica foi criada com a finalidade de encapsular as funções de controle: a classe Controle. Uma outra característica importante do método são os elementos que, como foi dito, podem ter características diferentes, porém com uma estrutura comum. Por isto, foi criada a classe Element. A classe Element se especializará para cada tipo de elemento finito, formando uma grande hierarquia. As operações com os arquivos de entrada e saída também devem ser encapsuladas. Para isto, criamos a classe TakeFile.

As operações matemáticas com vetores e matrizes, que são realizadas durante o processamento, estão todas presentes em classes preexistentes no GOPAC, não tendo que ser implementadas. Outra parte importante é o armazenamento dinâmico das informações em memória. Teríamos que identificar uma estrutura, dentro das várias estruturas já existentes, com a finalidade de verificar qual a mais eficiente. Uma base para a construção destas estruturas também já era existente no GOPAC [Sil94], mas, para seu uso, é necessário especializá-las.

3.3 Analisando a estrutura existente:

Ao iniciarmos a construção do framework, decidimos aproveitar um software já existente no laboratório: o Softwave [Lom97]. Esse software foi construído para resolver problemas em alta freqüência, utilizando o método de Elementos Finitos e o método de Equações Integrais. A idéia era extrair a experiência nele embutida, facilitando nosso trabalho e seguindo o espírito de construção de frameworks: basear-se na experiência acumulada com sistemas construídos anteriormente, como visto no capítulo 2. Iniciamos um estudo desse software. A ferramenta escolhida para documentação, Rational Rose, que segue os padrões da UML, possui um utilitário, que produz, a partir do

código existente, um diagrama de classes. Conseguimos então um diagrama atualizado do Softwave e começamos a analisar a hierarquia e funções das suas classes.

Na figura 3.3.1 podemos observar a classe GenericTree extraída do Softwave. Essa classe foi a base para todas as árvores utilizadas no sistema. Ela tem a função de ser um container, isto é, uma estrutura para armazenamento e gerenciamento dinâmico de dados. Essa classe vai se especializando nas distintas árvores que o sistema pode precisar. Esse container será necessário para o framework, visto que, pelo menos, elementos e nós têm que ter suas respectivas árvores. Podemos observar que essa árvore traz as funções básicas: o construtor GenericTree(), o destrutor ~GenericTree(), a função Initialize() que é utilizada para alocar o número de espaços necessários na árvore, que está sendo trabalhada, a função Add() que simplesmente guarda nesses espaços as suas informações, a função Get() que retorna os dados de uma devida posição na árvore, a função DoToAll() que retorna o próximo item da árvore e a sobrecarga de um operador de colchetes, Operator[](), para a identificação das posições nessa árvore.

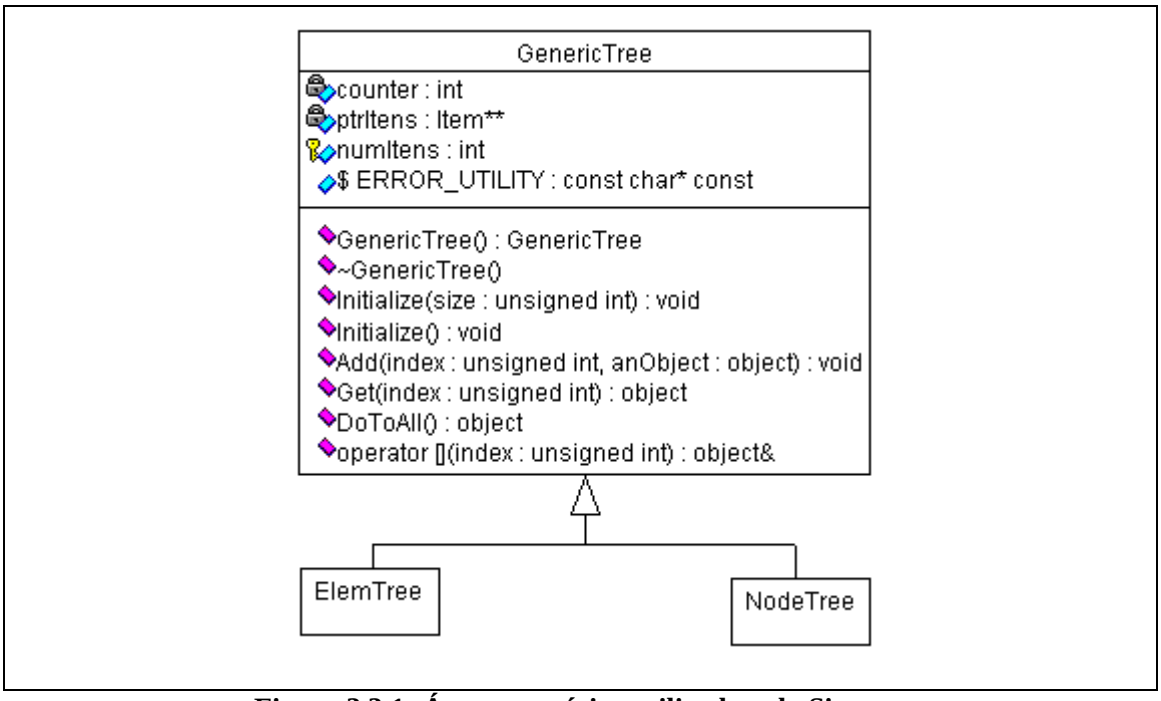

Figura 3.3.1: Árvore genérica utilizada pelo Sistema

Nas figuras 3.3.2 e 3.3.3 estão as classes Matrix e Vector que encapsulam todas as operações com matrizes e vetores. As classes Matrixex e Vectorex são classes que trabalham com matrizes e vetores complexos. O framework resolverá problemas estáticos, que geram matrizes e vetores reais, e quase-estáticos em regime permanente, que geram matrizes complexas. Nessas figuras não estão sendo mostrados todos os métodos dessas classes, pois retiramos as funções redundantes, construtores sobrecarregados, etc, de maneira a facilitar o seu entendimento.

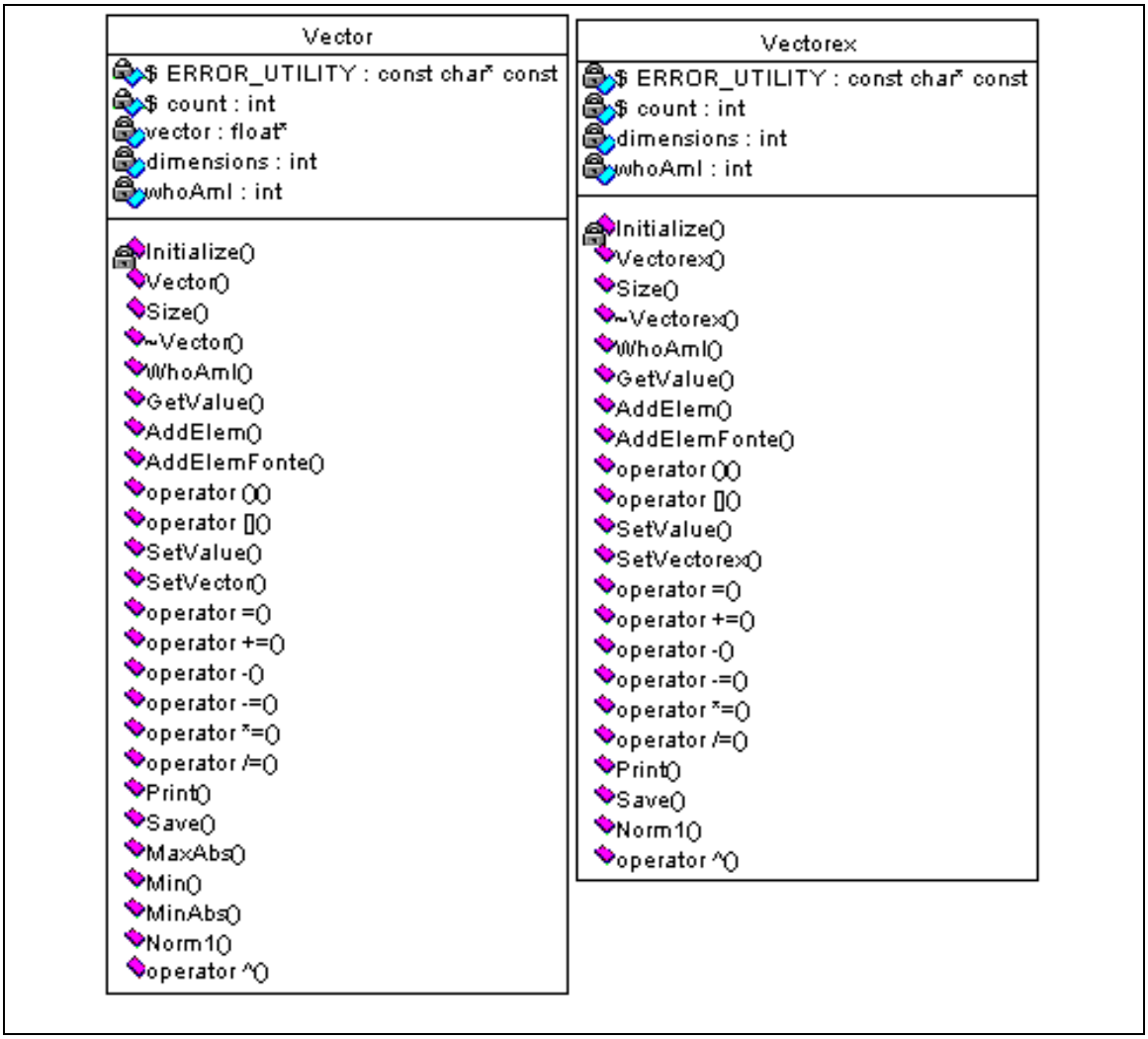

Figura 3.3.2: Classes utilizadas para o tratamento de Vetores e Matrizes

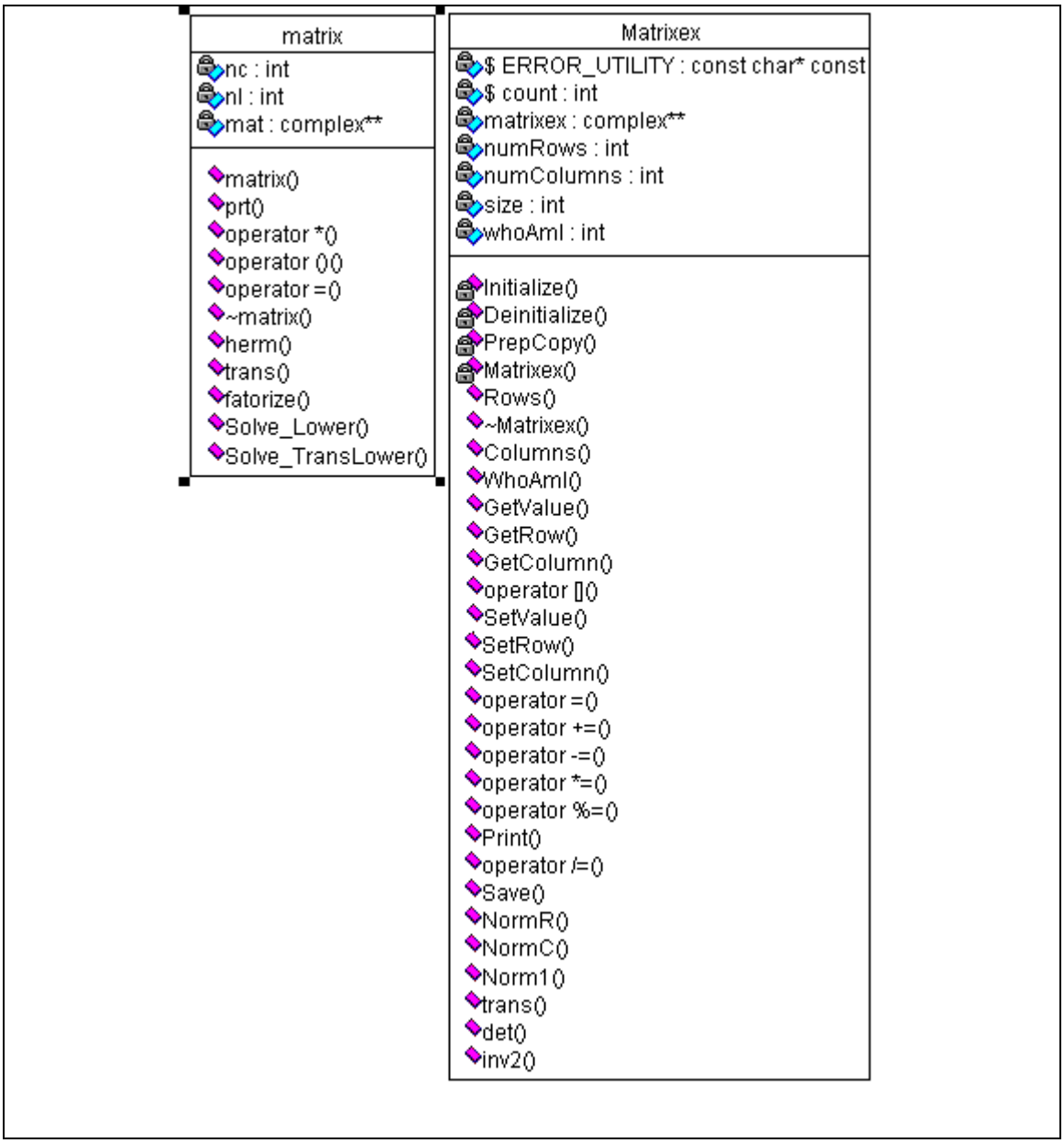

Figura 3.3.3: Classes utilizadas para o tratamento de Vetores e Matrizes

Na figura 3.3.4 observamos as classes MatrixSparse e ComplexMatrixSparse. Elas são utilizadas no desenvolvimento do método de Elementos Finitos, pois este gera matrizes esparsas.

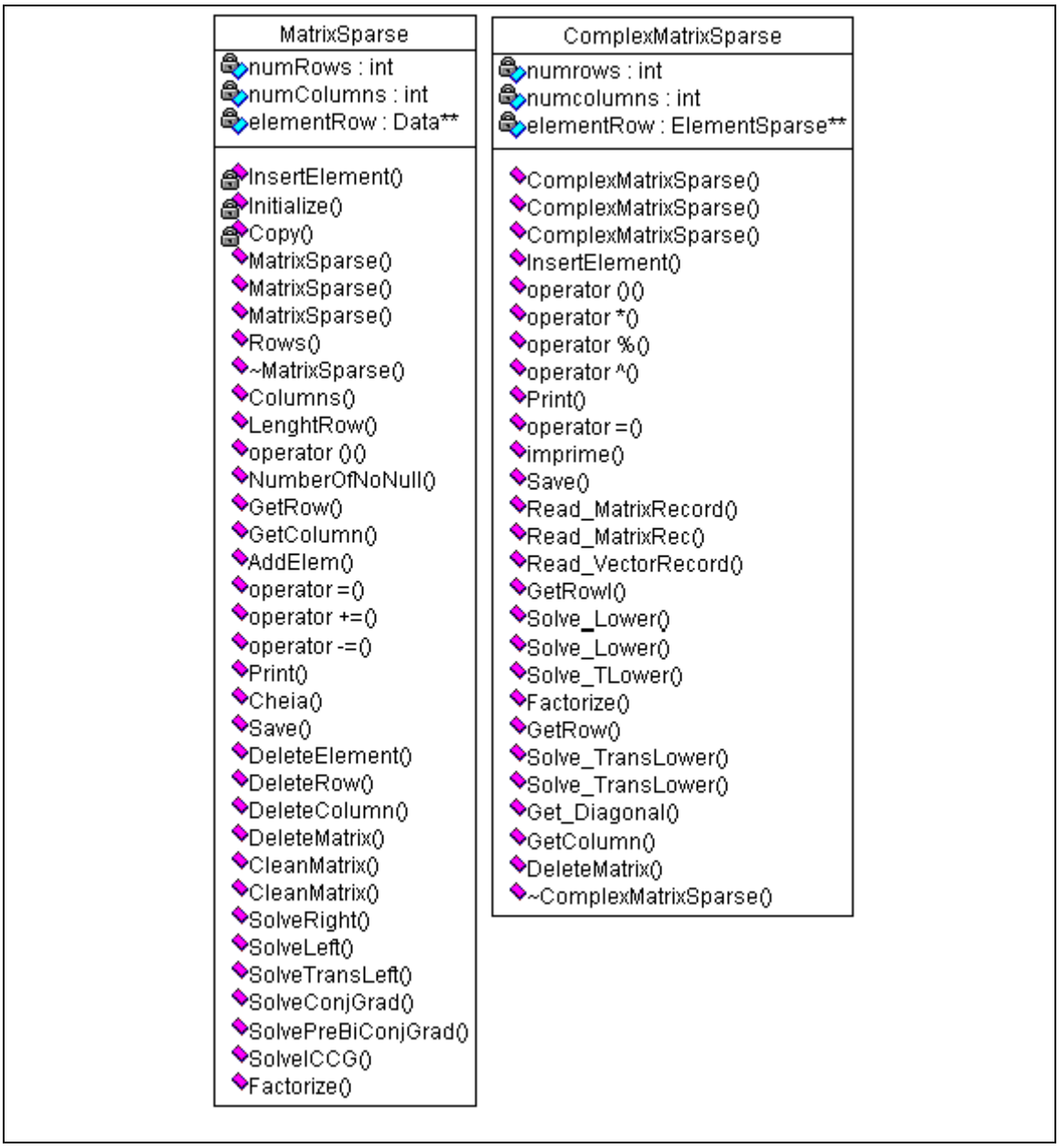

Figura 3.3.4: Classes que tratam a esparsidade das matrizes

A figura 3.3.5 mostra a classe Element, original do Softwave, junto com suas funções e hierarquia. Esta é uma classe de grande importância para o framework. O framework solucionará problemas em uma e duas dimensões, utilizando para isso elementos de linha, triangulares e quadrangulares. Analisando a hierarquia apresentada pelo diagrama, podemos observar que não são tratados problemas de uma dimensão e, para problemas de duas dimensões, existem elementos que não serão tratados nesse primeiro trabalho.

As classes Element_2d_qua, Element_2d_tri, Element_2d_tri_2_ord são utilizadas pelo método dos Elementos Finitos e as demais pelo método de Equações Integrais, sendo essas dispensáveis. Neste momento para o framework teremos que implementar também Element_2d_qua_2_ord e os elementos referentes aos problemas de uma dimensão.

Quanto às funções da classe Element observamos que muitas não são necessárias para o nosso problema. As funções relativas ao método de Equações Integrais e interfaces gráficas estariam sobrecarregando o framework. No alto da hierarquia estão sendo colocadas funções que não são compartilhadas por todos elementos, além disto, em um bom projeto, a interface deve ser claramente separada do núcleo de cálculo do sistema. Portanto, as funções de interface devem ser eliminadas deste nível. Para o framework, a classe Element deve ser a mais genérica possível, contendo apenas aquilo que for compartilhado por todos os elementos, o que não acontece com esta classe.

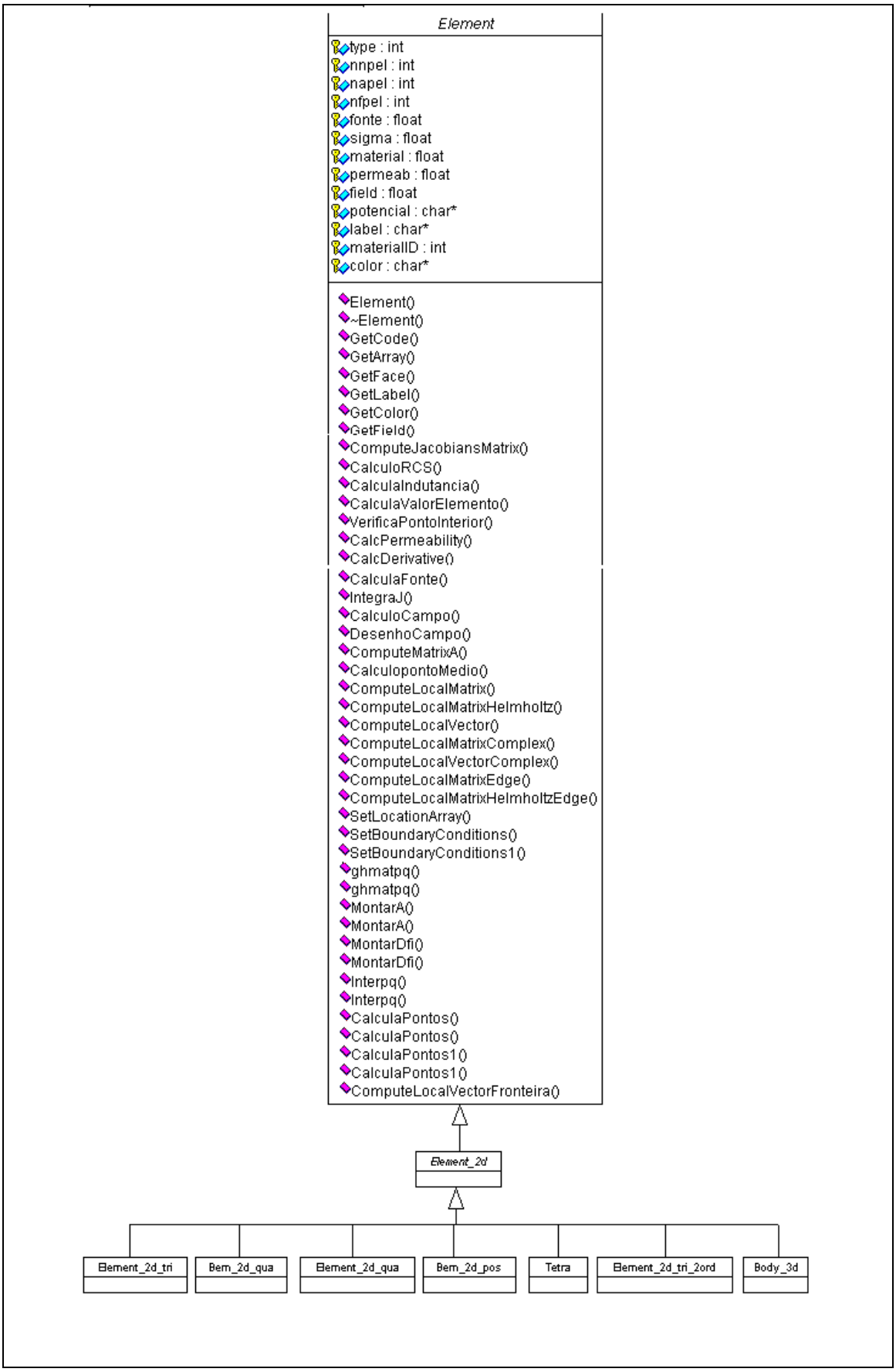

Figura 3.3.5: Hierarquia da classe Elemento no programa Softwave

O Softwave possui programação para Microsoft Windows e utiliza interfaces gráficas. O framework não terá interfaces gráficas e deverá possibilitar a mudança entre ambientes, não podendo, então, ser orientado para um tipo específico de plataforma. Os frameworks possuem o controle da manipulação de todas as suas classes e fluxo automático. O Softwave possui vários fluxos definidos pelo usuário.

Depois de analisar todas essas características, concluímos que o melhor caminho para a geração do framework seria a implementação de uma estrutura toda nova, onde seriam aproveitadas algumas classes, funções e formas de implementação do Softwave. Com essa estratégia ganharíamos tempo através do reaproveitamento da parcela de código mais encapsulada do sistema. Também, como apresentado no capítulo 2, esta é uma das maneiras que pode ser utilizada na implementação de um framework: utilizar implementações anteriores, generalizando-as, para que possam ser reutilizadas.

3.4 O projeto inicial do framework

A figura 3.4.1 representa o primeiro diagrama de classes do framework. Possui, além das classes do Softwave, as classes necessárias para a implementação da estrutura básica de um framework que serão programadas. A classe Controle é responsável pelo fluxo do framework, possuindo uma referência para todas as classes que têm que ser manipuladas diretamente. O construtor dessa classe gerencia todas as atividades do framework. A classe TakeFile foi criada com o objetivo de encapsular todas as operações com arquivos. Inicialmente todas as suas funções foram reaproveitadas do Softwave.

A hierarquia da classe Element e suas respectivas funções também foram reaproveitadas do Softwave, apesar das questões apontadas no item anterior. Ganhamos tempo em não ter que implementar o conteúdo das funções e ganhamos conhecimento na programação que já estava sendo utilizada pelo Grupo de Otimização e Projetos Assistidos por Computador – GOPAC - há

bastante tempo. Porém, sabíamos que modificações seriam necessárias para adaptar esta classe ao espírito do framework.

A classe Interface é uma classe que pretende encapsular toda a parte de comunicação do framework com o meio externo. Para o framework ela será bem simples, visto que, não é objetivo do framework nenhuma implementação gráfica. Essa classe abre caminho para que futuras implementações e aplicações que utilizem o framework, encapsulem suas interfaces gráficas.

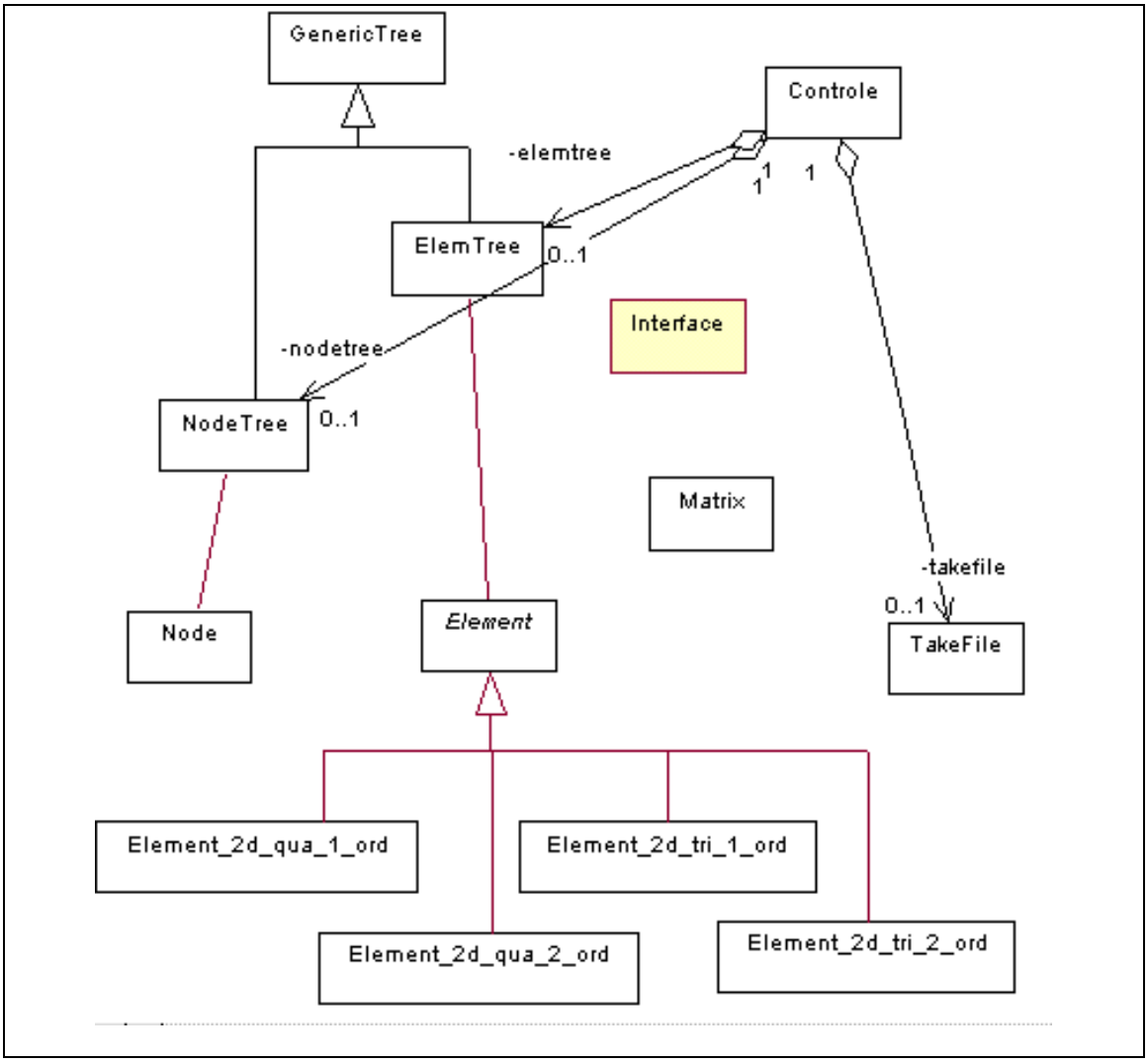

Figura 3.4.1: Primeiro diagrama de classes do framework

3.5 O primeiro minissistema

Obedecendo à técnica mostrada no capítulo 2, chegarmos de minissistema em minissistema a todo o sistema. Determinamos, então, qual seria o nosso primeiro minissistema e criamos o nosso primeiro objetivo: implementar as classes de forma que o futuro framework resolva problemas estáticos, utilizando elementos quadrangulares de primeira ordem. Essa técnica se encaixa perfeitamente aos frameworks, visto que o fluxo para a solução de um problema é praticamente o mesmo para a solução de outros problemas. Depois de concluirmos o primeiro minissistema, os outros não terão mais que ter o seu fluxo codificado: este simplesmente será herdado.

Durante a implementação do primeiro minissistema foram feitas várias mudanças para adaptarmos o código às especificações de um código totalmente orientado a objetos. As classes reaproveitadas possuíam muitos atributos públicos. Esses atributos podem ser manipulados sem a utilização de funções adequadas: facilitam a programação, mas diminuem o encapsulamento do programa. Com essa característica nós perdemos o controle de quem está fazendo o que com os atributos. Todos os atributos públicos foram modificados para privados e as suas respectivas funções de manipulação foram criadas. As classes também possuíam muitos membros “friends”. Esses facilitam a programação, permitem que outras classes consigam manipular as funções da classe amiga, mas também diminuem o encapsulamento e controle de acessos. O membros friends foram retirados para diminuir o acoplamento entre as classes.

O trabalho para a geração do primeiro minissistema demorou, aproximadamente três meses, e o grande aprendizado dessa fase foi: adaptações podem gerar modificações em cascata, especialmente se o código original não foi projetado com a intenção de reutilização como princípio básico. Foram descobertos aspectos interessantes: a forma utilizada para leitura do arquivo neutro e a programação utilizada para manipulação das árvores que armazenam os nós e elementos são extremamente eficientes.

Outro aspecto importante durante a construção dos minissistemas são os testes. Começamos com testes básicos e então com a evolução do framework,

testes mais complexos. A figura 3.5.1 ilustra o primeiro teste, criamos um arquivo neutro contendo um problema eletrostático. Esse problema possui um elemento quadrangular de primeira ordem, duas condições de contorno e está preenchido com ar, ficando fácil para calcularmos o resultado e então começarmos a testar o minissistema. O elemento possui lados iguais. A variação do potencial é linear de 0 a 1 v. Portanto, no meio deste intervalo, teremos a metade do potencial, ou seja, 0,5 v.

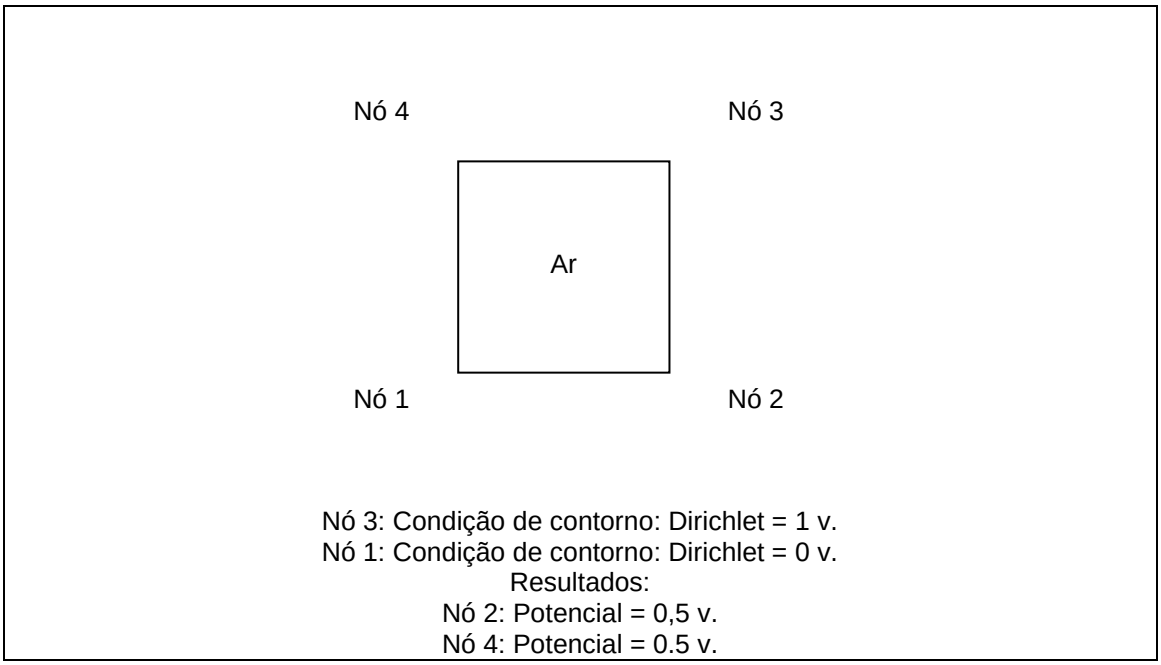

Figura 3.5.1: Representação do elemento utilizado para teste do primeiro minissistema

3.6 O segundo minissistema

Com o primeiro minissistema funcionando, partimos para a geração do segundo minissistema: integrar ao futuro framework a solução de problemas estáticos, utilizando elementos triangulares de primeira ordem.

Começamos a observar nessa fase vantagens decorrentes do uso dos frameworks. Para essa implementação, algumas preocupações não foram mais necessárias. O fluxo para solução dos problemas foi herdado do primeiro minissistema, restando somente a responsabilidade de implementar as diferenças ocasionadas pelos elementos triangulares.

Esse desenvolvimento foi mais rápido, pois a classe que controla as operações efetuadas com os arquivos já estava funcionando, o fluxo estava pronto, sendo tudo isso herdado pelo minissistema, restando então a parte da formulação para ser resolvida. O desenvolvimento foi tão simples que decidimos integrar ao segundo minissistema os elementos triangulares de segunda ordem. Tendo muita coisa sido herdada do Softwave e também do framework, que estava tomando forma, o desenvolvimento foi efetuado rapidamente.

Como teste utilizamos inicialmente o mesmo problema do minissistema anterior. As figuras 3.6.1 e 3.6.2 representam dois problemas eletrostáticos. No primeiro, o elemento quadrangular de primeira ordem foi dividido em dois elementos triangulares de primeira ordem. No segundo o elemento quadrangular de primeira ordem foi dividido em dois elementos triangulares de segunda ordem. Utilizamos, novamente, as mesmas condições de contorno, ficando fácil para calcularmos os resultados que, nos nós coincidentes, correspondem aos mesmos resultados mostrados anteriormente.

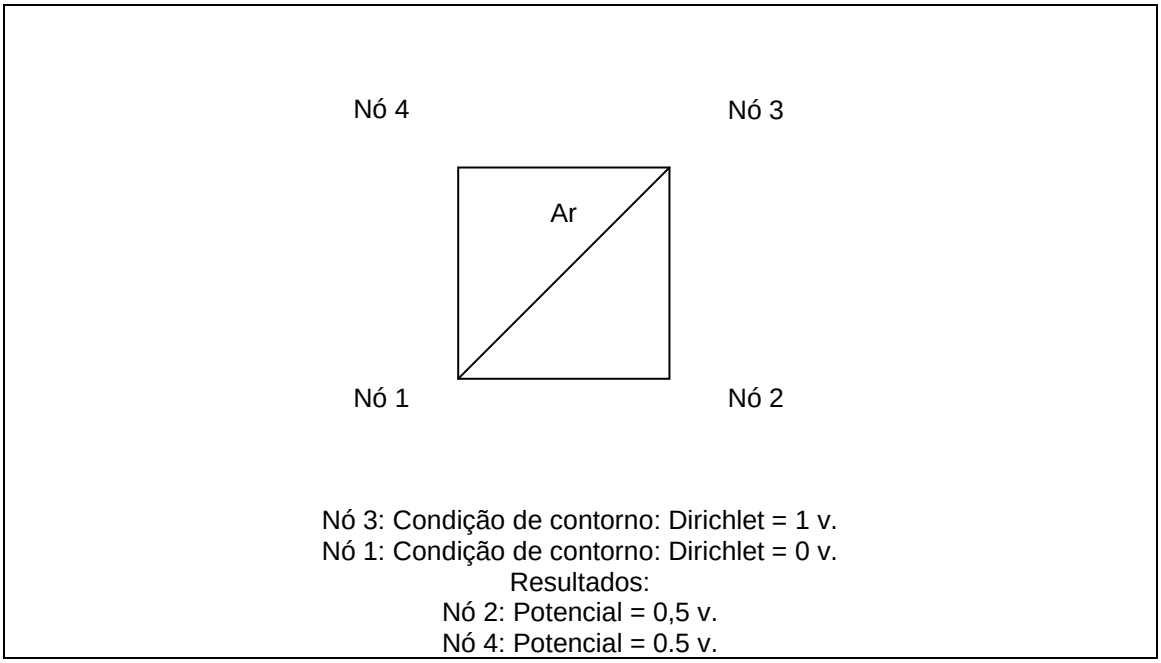

Figura 3.6.1: Representação dos dois elementos triangulares de primeira ordem utilizados para os teste do segundo minissistema

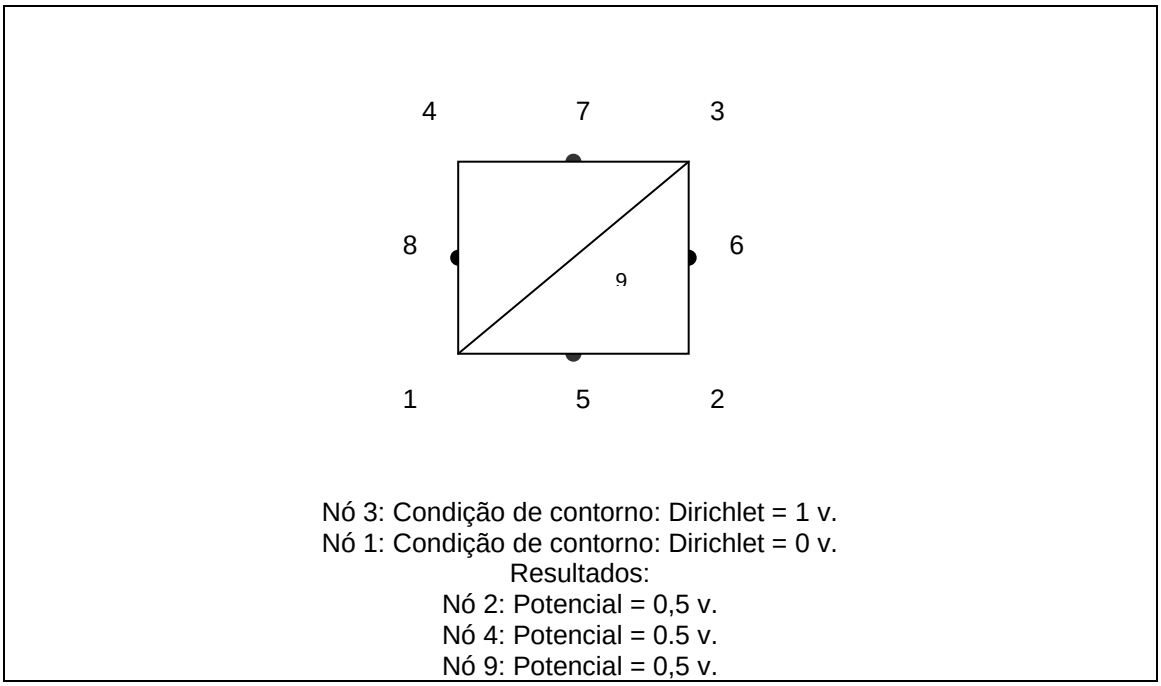

Figura 3.6.2: Representação dos elementos triangulares de segunda ordem utilizados para o teste do segundo minissistema

3.7 O terceiro minissistema

Com o segundo minissistema funcionando, partimos então para a geração do terceiro minissistema: integrar ao futuro framework a solução de problemas estáticos, utilizando elementos quadrangulares de segunda ordem.

Os elementos quadrangulares de segunda ordem foram totalmente desenvolvidos. Herdamos, fora as características do framework, apenas aquelas funções iguais às funções dos elementos quadrangulares de primeira ordem. Nessa etapa do trabalho, começamos a identificar problemas na hierarquia das classes do framework. Vejamos a hierarquia que está sendo apresentada na figura 3.7.1: da forma como essa hierarquia foi implementada, começamos a perceber duplicidade de código. Todas as classes abaixo de Element_2d são concretas, portanto, todas as funções virtuais da classe Element_2d terão que ser definidas nessas classes. Vejamos, por exemplo, as funções ComputeLocalMatrix() e ComputeLocalVector().

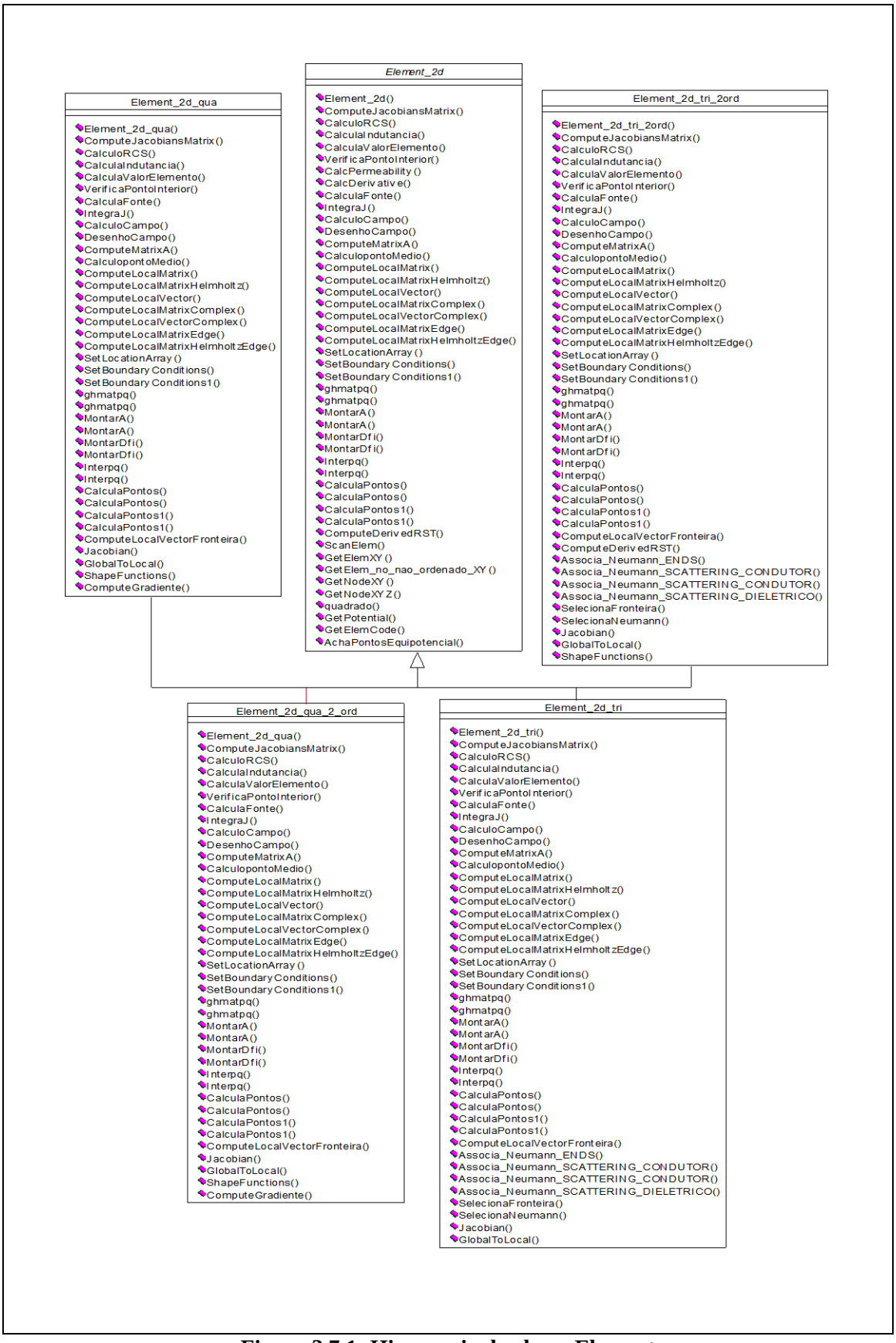

Figura 3.7.1: Hierarquia da classe Element

Essas funções possuem a mesma definição para elementos de mesma forma, independente do seu número de nós. Esse problema pode ser resolvido com uma alteração na estrutura das classes.

A nova estrutura, figura 3.7.2, para a hierarquia de classes resolve completamente o problema da duplicidade de código [Mai00]. As classes Element_2d_Qua e Element_2d_tri possuem as definições comuns para as suas respectivas especializações. É importante explicitar que nas classes superiores devem estar definições das funções comuns a todas as classes inferiores, pois isso é primordial para evitar a duplicidade de código. Aproveitamos também para incluir nessa hierarquia os elementos de uma dimensão que também serão trabalhados pelo framework. Como exemplo para demonstrar a viabilidade da hierarquia, analisemos a função GetElemXY(): essa função tem o objetivo de retornar a coordenada dos nós. Tem definições diferentes para problemas de dimensões diferentes. Dessa forma sua definição deve estar no nível mais alto de cada dimensão, pois, será a mesma para todo o restante das especializações. Esse raciocínio deve ser utilizado para a definição de elementos de terceira dimensão, já que a lógica é a mesma.

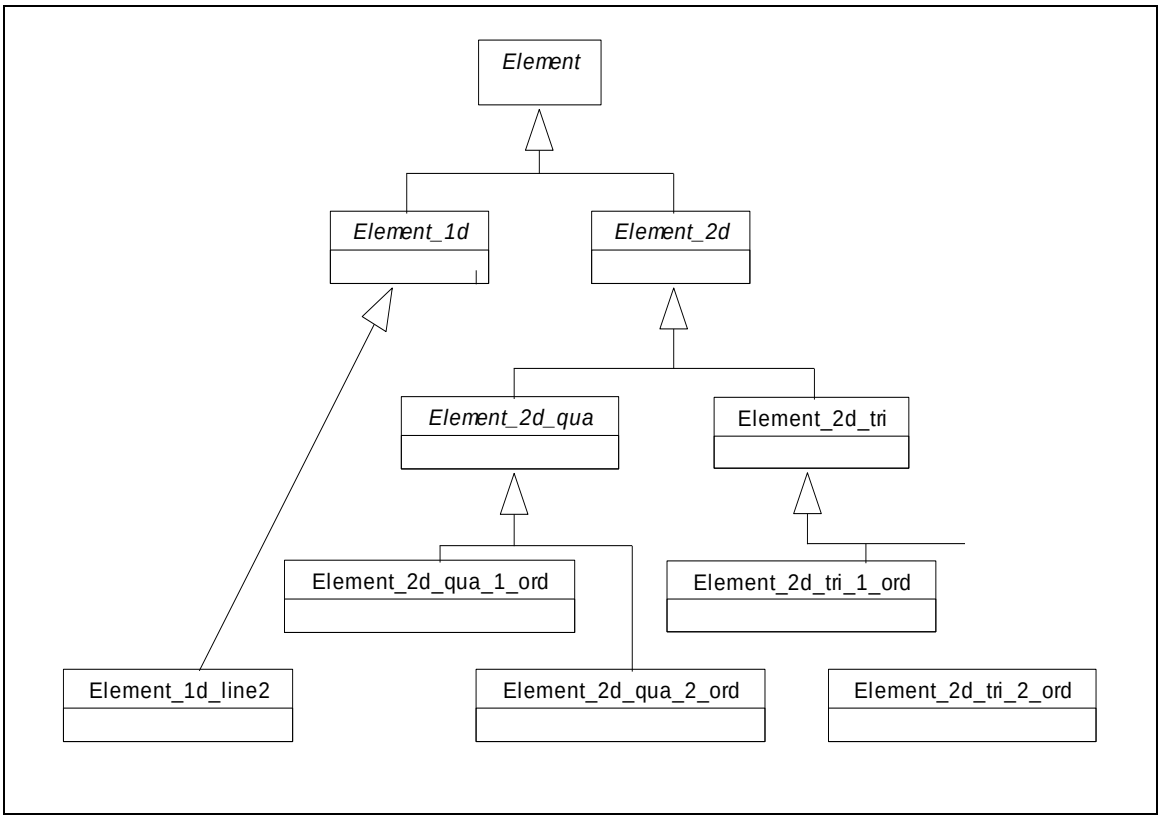

Figura 3.7.2: Nova estrutura para a classe Element

Com o terceiro minissistema funcionando, começamos a analisar as funções existentes nas classes derivadas da classe Elemento, figura 3.7.3 Nesse momento, o futuro framework estava resolvendo problemas estáticos, utilizando elementos quadrangulares e triangulares. Essa análise é importante para verificarmos a necessidade de outras generalizações, ou seja, criação, ou não, de outras classes.

O atributo fonte, que representa as influências das fontes elétricas ou magnéticas presente na superfície do problema, não é uma responsabilidade de Element. A partir do momento que pensamos na implementação de problemas mais complexos, esse atributo passa a ter suas próprias responsabilidades, o que nos faz implementar uma classe distinta: Fonte. O atributo Material, que representa o tipo de material presente no elemento, também não é uma responsabilidade do elemento. Existem problemas com vários tipos de materiais e esses materiais podem ser lineares ou não lineares, isotrópicos ou anisotrópicos, etc, tendo características as mais variadas possíveis. Assim, esse atributo adquire suas próprias responsabilidades, o que nos faz implementar uma classe distinta: Materiais. Os atributos Sigma e Permeab, que significam a condutividade e a permeabilidade do meio, representando propriedades de comportamento do material, pertencem a classe Material, sendo naturalmente transferidos para lá. O atributo Color diz respeito à visualização dos elementos, o que não é objetivo do framework.

Observando, agora, as funções da classe Element podemos ver que essa está bem sobrecarregada. Funções como GetFace(), GetColor(),GetRcs(), CalculaIndutância(), VerificaPontoMedio() são funções que também fogem do nosso objetivo e, se analisarmos mais profundamente, poderemos até mesmo migrar essas funções para outras classes, deixando na classe Element apenas funções de sua única responsabilidade. A classe Element terá uma referência para essas classes, utilizando essa referência para acessar o que for necessário. Podemos observar que a necessidade de mudanças mostra que a classe está pouco genérica. O objetivo é deixar em Element apenas o que é responsabilidade do Element. A classe que está sendo trabalhada é a mais

genérica, devendo conter somente o que pertence a todos os elementos. As diferenças existentes devem ficar nas respectivas especializações.

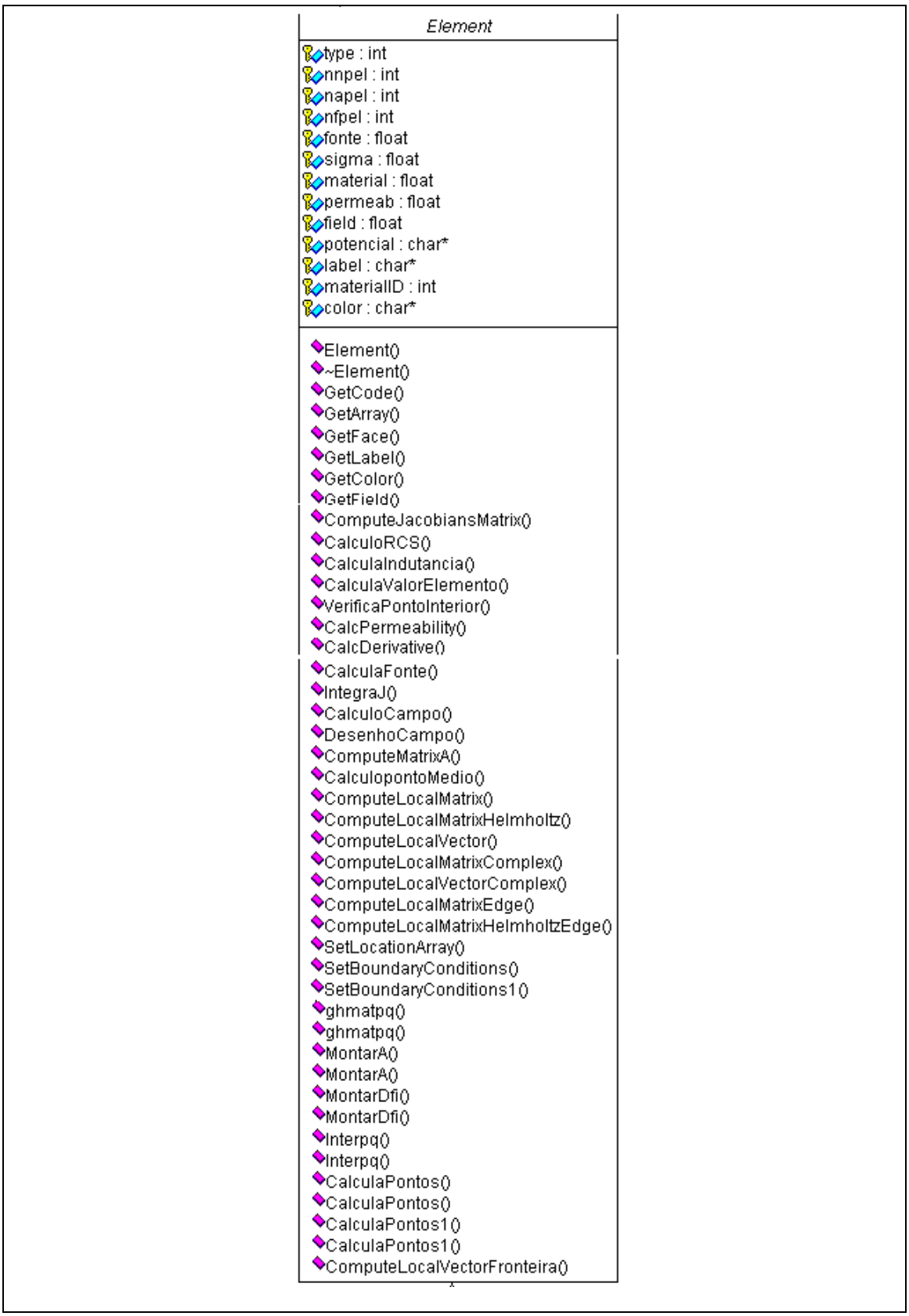

Figura 3.7.3: Primeira modificação da classe Element

Feitas essas mudanças, observemos novamente a classe Element na figura 3.7.4. Podemos notar que a classe estava realmente bem carregada. A classe agora está mais simples e de fácil entendimento. É importante começarmos a perceber como as generalizações exigem conhecimento, tornando mais simples o entendimento das classes, ao mesmo tempo que, com a criação de novas classes, a programação fica bem mais complexa.

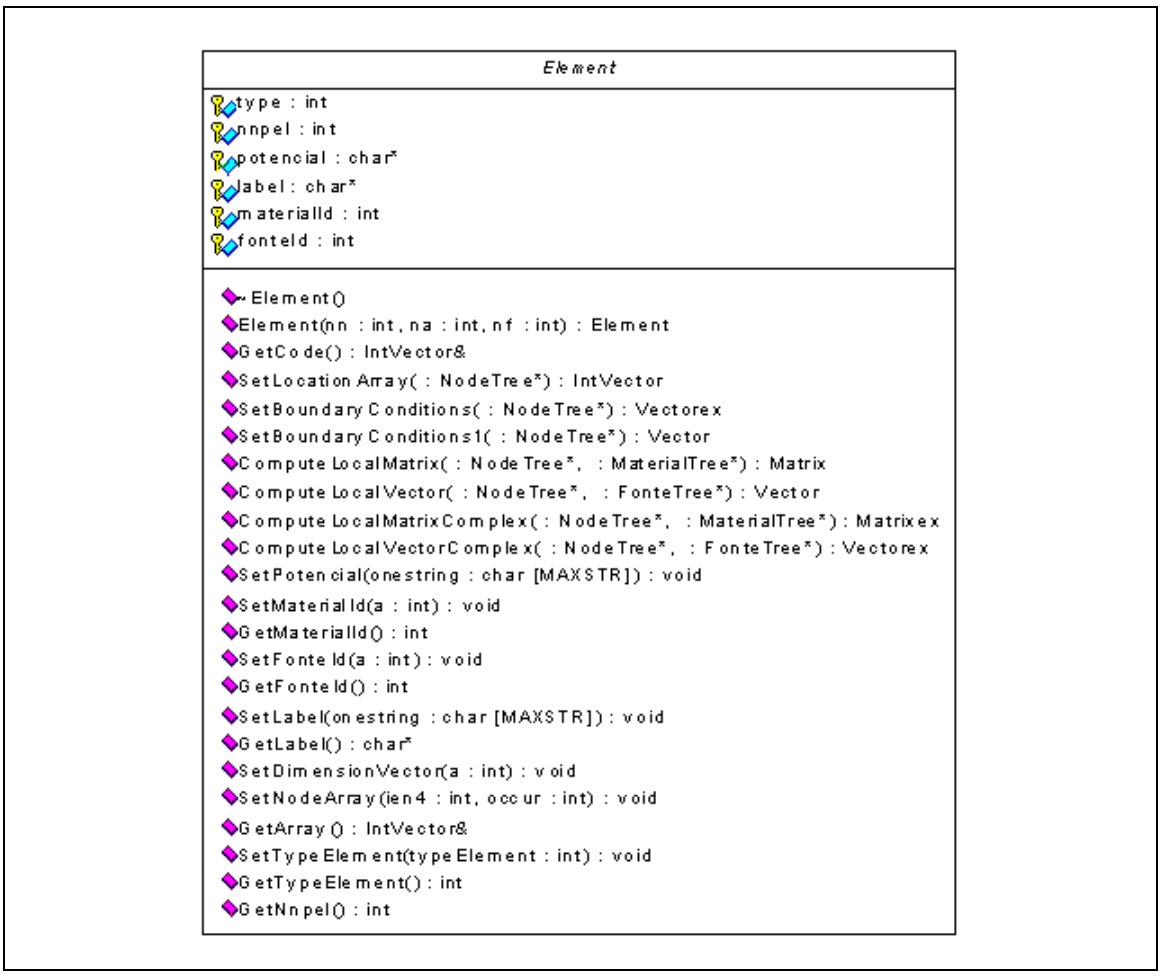

Figura 3.7.4: Segunda modificação da classe Element

Como teste criamos um arquivo neutro contendo um problema eletrostático. Criamos mais nós no elemento quadrangular de primeira ordem, transformando-o em um elemento quadrangular de segunda ordem. Utilizamos novamente as mesmas condições de contorno, ficando fácil para calcularmos o resultado e então testar o minissistema. A figura 3.7.5 ilustra o problema utilizado para o teste.

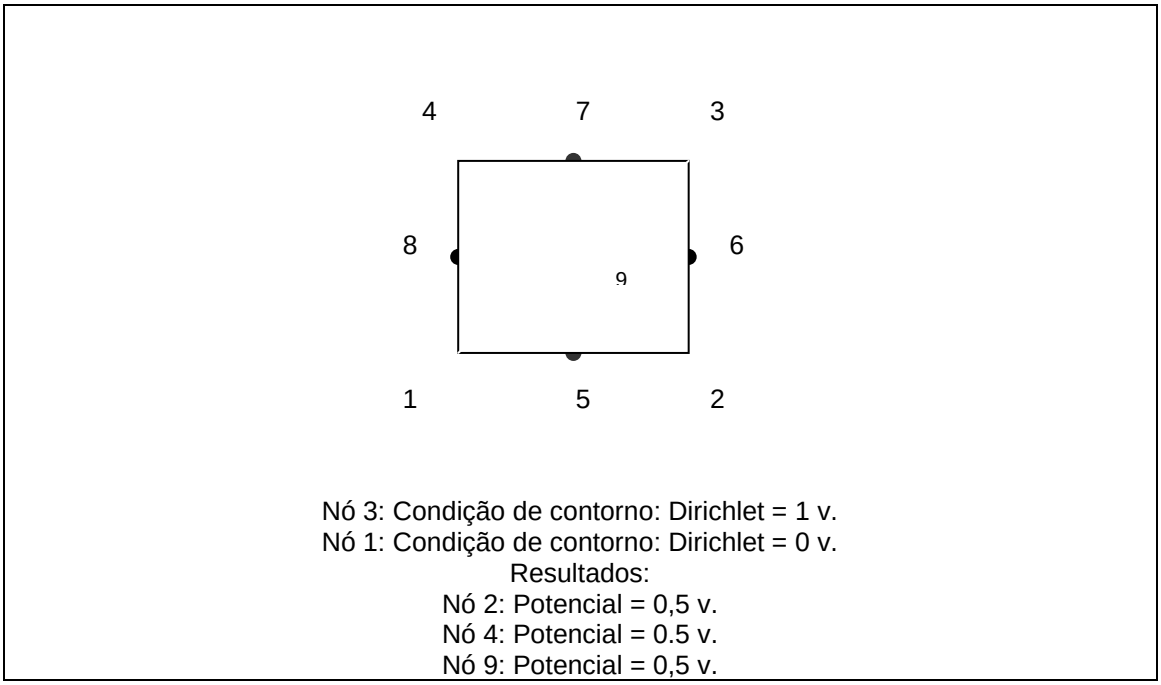

Figura 3.7.5: Elemento quadrangular de segunda ordem utilizado para os testes do terceiro minissistema

Neste momento incluímos também, como teste, um problema magnetostático. O objetivo era verificar, também, a manipulação das contribuições geradas por uma fonte de corrente. A figura 3.7.6 representa a geometria desse problema. Mostra um condutor percorrido por uma corrente de 10 A, circunferência menor, de raio igual a 1 metro e uma circunferência maior, raio igual a 10 metros envolvida por ar, limitando o problema. A circunferência maior simboliza o infinito através da condição de contorno imposta, dirichlet 0. O framework calcula o campo magnético em vários pontos no interior dessa circunferência.

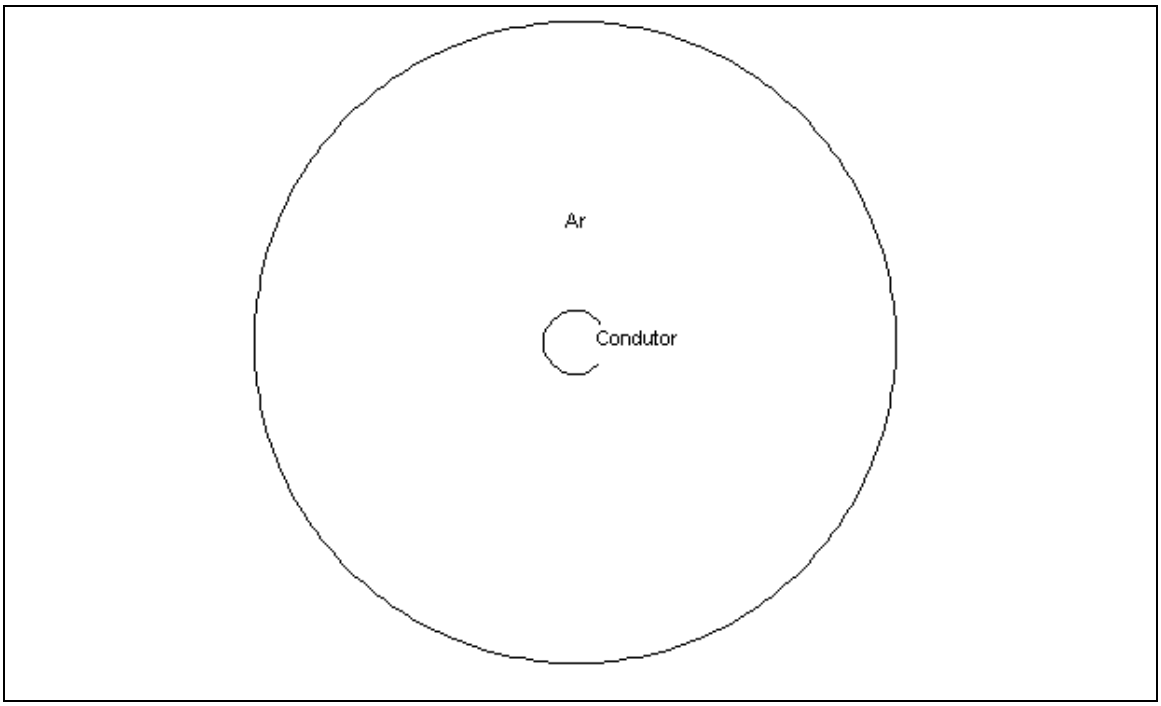

Figura 3.7.6: Geometria de um problema magnetostático

A figura 3.7.7 apresenta resultados obtidos pelo framework e através dos cálculos analíticos.

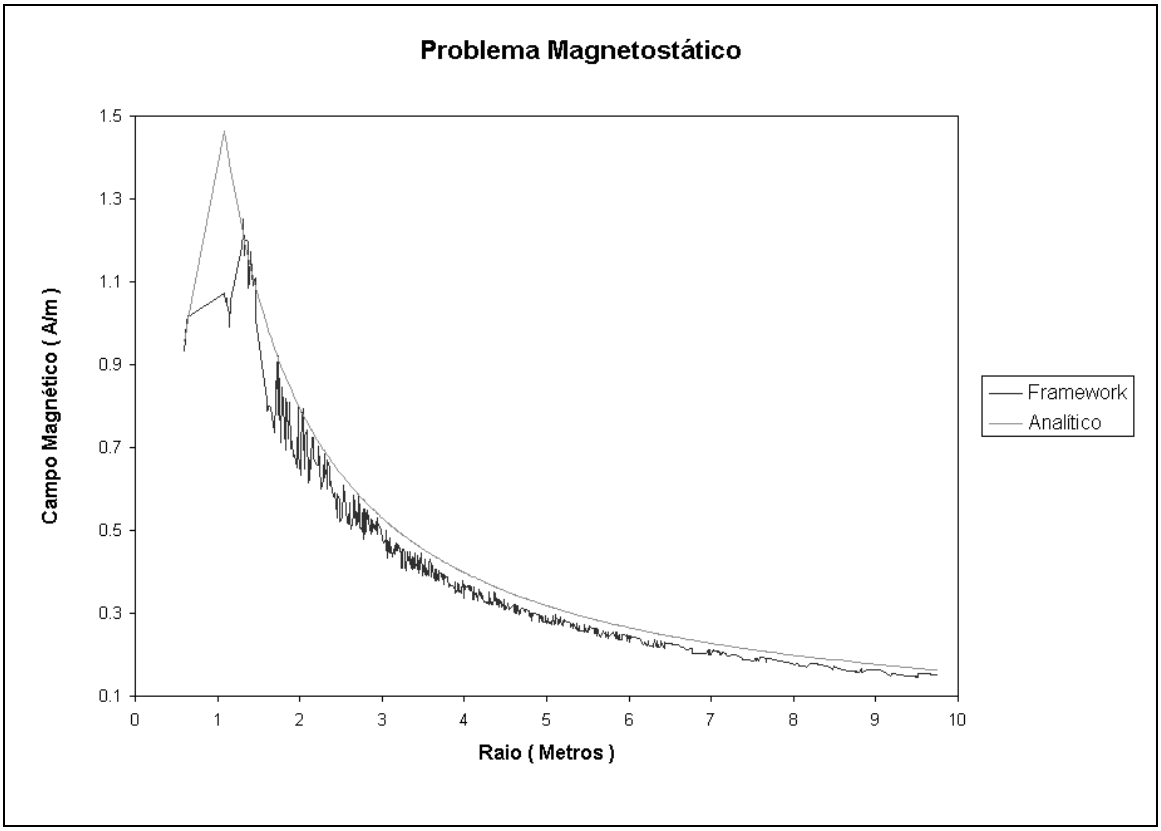

Figura 3.7.7: Comparação dos resultados calculados e obtidos pelo framework

3.8 O quarto minissistema

Com o terceiro minissistema funcionando, partimos para a geração do quarto minissistema: integrar ao framework a solução de problemas quase-estáticos, utilizando elementos triangulares de primeira ordem.

O código herdado do Softwave não trabalhava problemas quase estáticos, mas trabalhava problemas em alta freqüência, possuindo funções que utilizavam números complexos. Mais uma vez, verificamos a viabilidade dos frameworks: herdamos o fluxo de controle , herdamos toda a parte que já havia sido implementada para a leitura de arquivos, herdamos a montagem da matriz de contribuições que formam o sistema matricial que, resolvido, gera a solução do problema, etc. Implementamos somente a diferença entre problemas estáticos e quase-estáticos. No nosso caso, em que estamos trabalhando no domínio da freqüência, a diferença é o tratamento da parte imaginária das matrizes e vetores gerados. Como o framework já trabalhava todos os tipos de elementos propostos para duas dimensões, integramos ao quarto minissistema a solução de problemas utilizando todos os elementos do framework.

Terminada a implementação para a solução dos problemas quase-estáticos, começamos a observar que, com algumas mudanças, seria possível generalizar ainda mais a classe Element. As funções ComputeLocalMatrix() ComputeLocalMatrixComplex() possuem pequenas diferenças que podem ser encapsuladas em um único ComputeLocalMatrix(). A primeira mudança seria tratar as funções que tratam a matriz de contribuições de um elemento, Ke, e o vetor de contribuições de um elemento, Fe, como se fossem sempre números complexos. Logicamente para os problemas estáticos a parte imaginária seria sempre zero. A Segunda mudança seria para diferenciar as formulações de um e outro problema. Começando a estudar as formulações, vimos que existem diversos tipos de formulações, como, por exemplo, as formulações para problemas estáticos, para problemas quase-estáticos e para problemas em alta freqüência. Se todas essas formas para solução de problemas fossem ficar dentro da função que calcula a matriz Ke e o vetor Fe, matriz e vetor de

contribuições gerados para cada elemento, essas funções ficariam altamente sobrecarregadas e de difícil manutenção. Resolvemos, então, iniciar mais um nível de abstração no framework, criando uma classe que tratará todos os tipos de formulação: a classe Formulation. Observemos na figura 3.8.1 a proposta inicial para a hierarquia dessa classe. A hierarquia apresentada ainda é bem simples, contemplando só os tipos de problemas mais básicos: eletrostáticos, magnetostaticos e quase-estáticos para uma e duas dimensões.

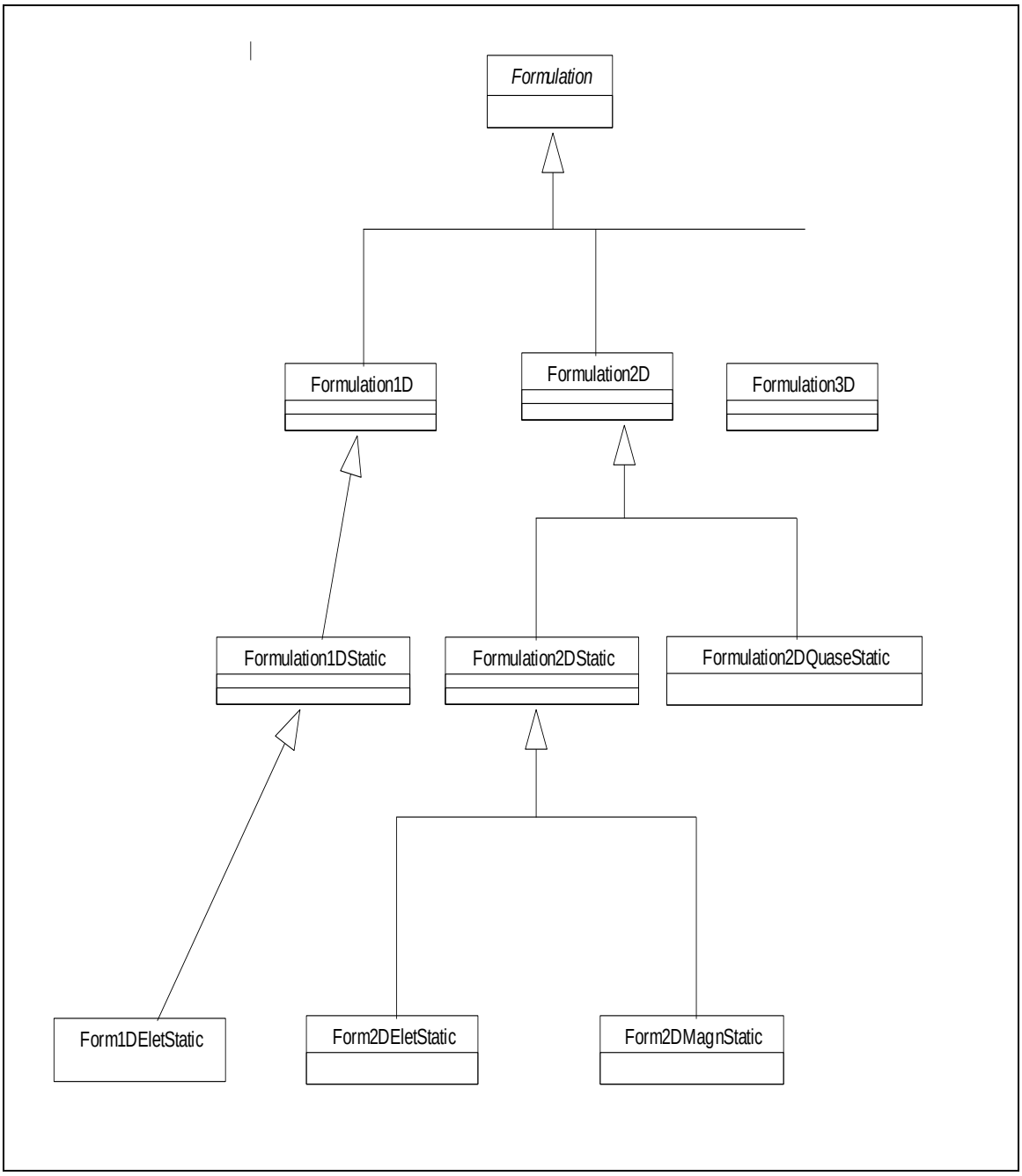

Figura 3.8.1: Hierarquia da classe Formulation

O framework utiliza o método de Elementos Finitos para resolver problemas de vários tipos, como mostrado na figura 3.8.1. Cada um desses problemas possui diferenças no cálculo da matriz e do vetor de contribuição dos elementos.

A classe Formulation passa a ter a responsabilidade de mostrar ao elemento como realizar esses diversos cálculos. Utilizamos para essa implementação uma técnica conhecida como duplo despacho [Bar94]. No início do processamento, durante a leitura do arquivo neutro, as informações sobre o tipo de problema, número de elementos, dimensões do problema etc, são armazenadas. No decorrer desse processo, os objetos, elementos, nós, etc, são gerados. Durante o processamento de cada elemento, o tipo de problema é checado e aí gerado um objeto para o tipo correto da formulação. No processamento da matriz e do vetor locais, são executadas somente as funções comuns a todos os problemas. Quando é chegado o momento do cálculo das contribuições, que depende da formulação empregada, o elemento faz o despacho de atribuições para a classe Formulation, informando os dados necessários, inclusive o seu próprio endereço. Como a Formulação já possui um objeto do tipo adequado, esse objeto através do endereço e dados necessários recebidos do elemento, indica ao elemento o que deve ser feito para resolver o tipo de problema trabalhado. Esta implementação será exemplificada com mais detalhes com os diagramas da seqüência deste capítulo.

Terminando a implementação do duplo despacho, atentamos para mais um detalhe no framework: a necessidade do atributo type na classe Element. Verificamos que a sua única função era a distinção dos tipos de elementos que iam ser trabalhados pelo framework, triangulares de primeira ordem, quadrangulares de segunda ordem, etc. Essa identificação não é necessária durante o processamento de cada elemento. A orientação a objetos introduz conceitos interessantes: os objetos criados sabem suas características, inclusive sua forma. Essas características são dadas pelo Polimorfismo [Koe97], dispensando o uso do atributo type. Da forma como estava implementada essa característica, o polimorfismo não estava sendo utilizado.

Depois de implementadas essas mudanças, pode-se observar a nova classe Element na figura 3.8.2.

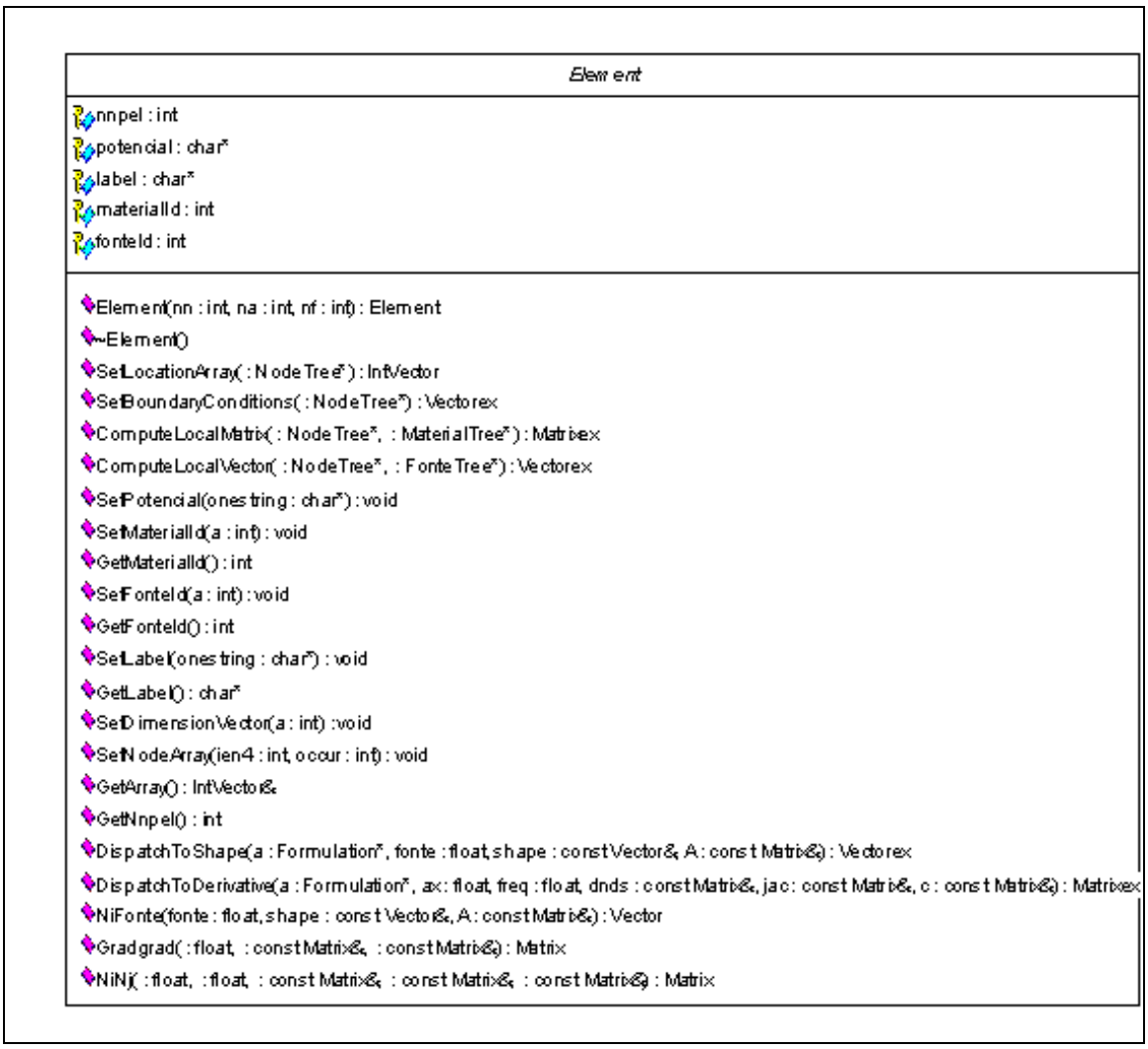

Figura 3.8.2: Terceira implementação da classe Element

As funções NiNj(..), Nifonte(..), Gradgrad(..), são funções que, gerenciadas pela classe Formulation, conseguem resolver os diferentes tipos de problemas propostos para o framework. A função NiNj(..) calcula contribuições do tipo $\int_{\Omega} \sigma \, Ni.Nj \;\; d\Omega$, onde σ, sigma, representa o coeficiente de condutividade do material e Ni e Nj são as funções de forma, sendo utilizada nos problemas quase-estáticos. A função Nifonte(..) calcula contribuições do tipo $\int_{\Omega} Ni.f \, d\Omega$, onde f é associado a uma fonte de atuação

no domínio (por exemplo a densidade de corrente elétrica J). A função Gradgrad(..) calcula as contribuições do tipo $\int_{\Omega} k \vec{\nabla} Ni . \vec{\nabla} Nj \; d\Omega$, onde k representa o coeficiente de permeabilidade do material presente nesse elemento e $\vec{\nabla} Ni . \vec{\nabla} Nj$ representa o gradiente das funções de forma. A classe Formulation mostra como essas funções devem ser combinadas para solucionar os problemas. A utilização dessas funções se dá através do duplo despacho.

Neste momento incluímos, como teste, um problema quase-estático. A figura 3.8.3 ilustra a geometria desse problema. Um condutor retangular percorrido por uma corrente imposta de $1x10^7$ A, de frequência 10 Hertz. Um bloco condutor de cobre, de 8 x 2 metros, sofre a indução de corrente. Estes materiais estão envolvidos por ar e limitados pelo quadrado maior, 10 x 10 metros.

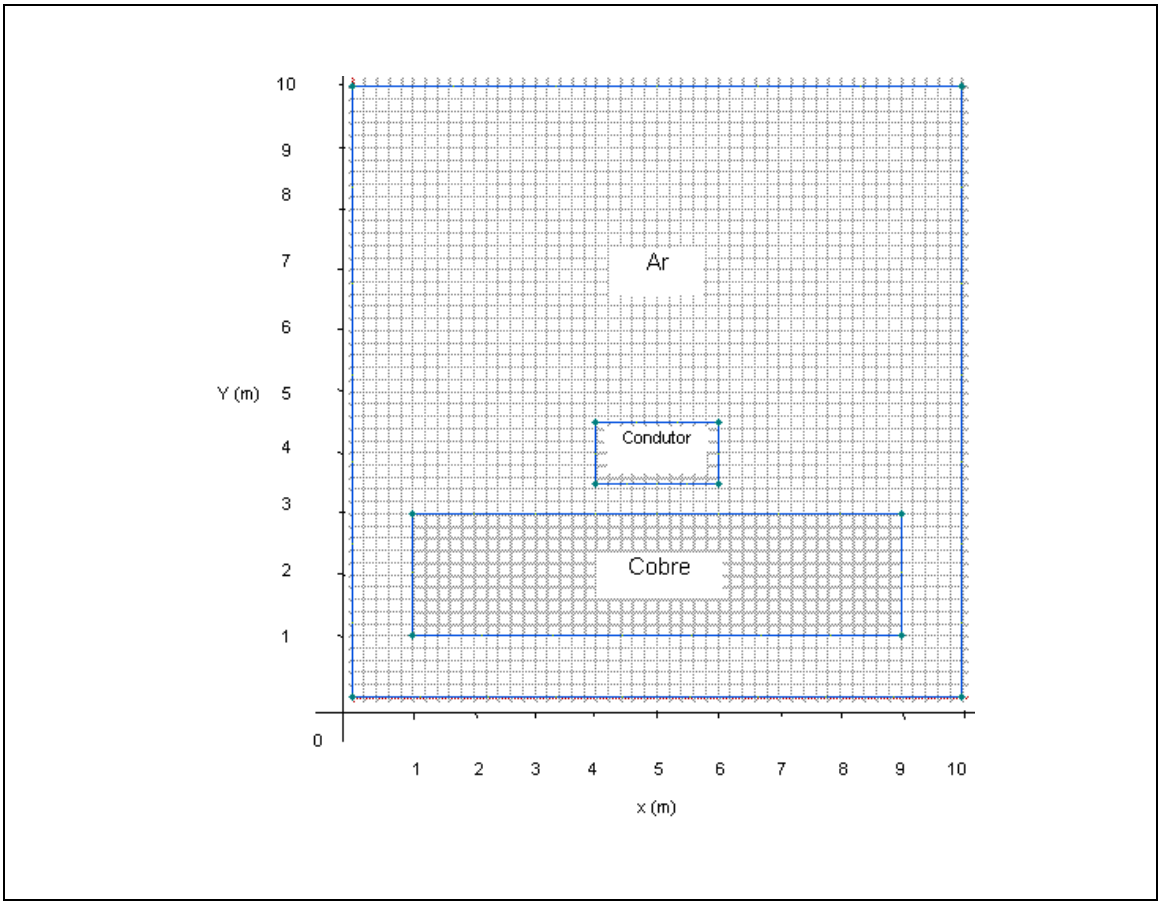

Figura 3.8.3: Geometria de um problema quase-estático

A figura 3.8.4 mostra a variação do módulo do campo magnético em uma linha vertical a geometria do problema, começando em (x=1,3 y=0,3) e terminando em (x=1,3 y=9,3). Foram utilizados para as soluções elementos quadrangulares e triangulares.

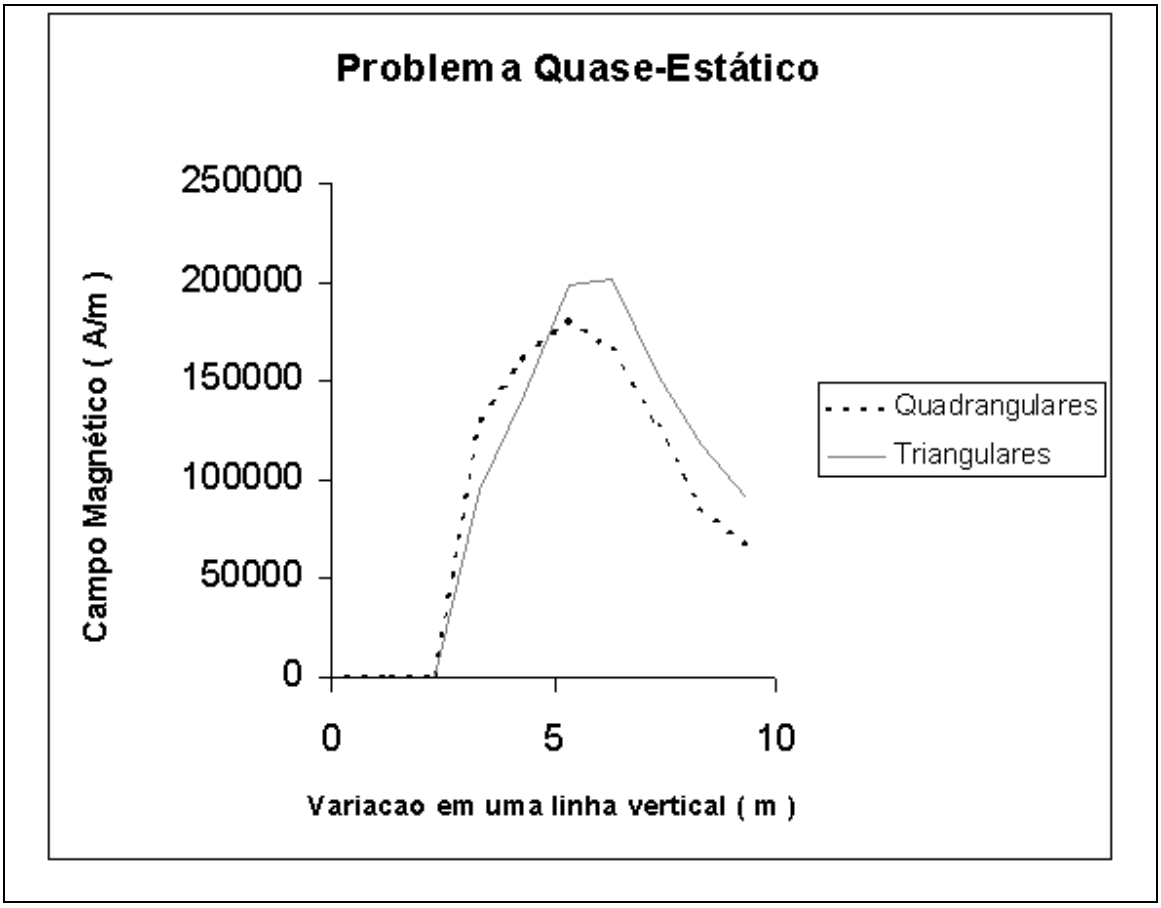

Figura 3.8.4: Comparação dos resultados obtidos com a utilização de elementos quadrangulares e triangulares

3.9 O quinto minissistema

Com o quarto minissistema pronto, começamos a implementação do quinto e último minissistema: integrar ao framework a solução de problemas estáticos, utilizando elementos de linha de primeira e segunda ordens.

Com o framework já quase todo estruturado, essa implementação foi a mais simples: herdamos fluxo de controle, herdamos leitura e escrita em arquivo, herdamos funções que são comuns a todos elementos e tivemos que implementar apenas o que era unicamente exclusivo de elementos lineares.

Como teste criamos dois arquivos neutros contendo problemas eletrostáticos. No primeiro utilizamos 6 elementos de linha de primeira ordem, no segundo, utilizamos na mesma região 3 elementos de linha de segunda ordem, criando de propósito, regiões iguais e com nós coincidentes nos dois arquivos. Utilizamos para os dois problemas as mesmas condições de contorno, ficando fácil para calcularmos o resultado e então testar o minissistema. A figura 3.9.1 ilustra os problemas.

Nó 7: Condição de contorno: Dirichlet = 1.2 v.
Nó 1: Condição de contorno: Dirichlet = 0 v.
Resultados:
Nó 2: Potencial = 0,2 v.
Nó 3: Potencial = 0.4 v.
Nó 4: Potencial = 0,6 v.
Nó 5: Potencial = 0.8 v.
Nó 6: Potencial = 1.0 v.

Figura 3.9.1: Representação dos elementos lineares utilizados para os testes do quinto minissistema

Nesse momento, o framework já estava resolvendo todos os problemas que tinham sido propostos. Havia ainda pequenos problemas que, aos poucos, seriam solucionados. Estávamos aptos, então, a prosseguir com uma parte muito importante: a documentação.

3.10 Estratégia de Análise, Projeto e Documentação

O Rational Rose, ferramenta de documentação da UML criada pela empresa Rational, possui vários tipos de diagrama que permitem uma boa documentação do software. Essa etapa do trabalho mostrará os diagramas utilizados. É importante deixar claro que a documentação estava sendo gerada desde o início do primeiro minissistema.

Para a geração do diagrama de classes do framework utilizamos novamente a engenharia reversa do Rational. A única dificuldade encontrada nessa etapa foi a quantidade de classes existentes no diagrama. Decidimos dividi-lo em blocos, correspondentes aos módulos do sistema. Assim, criamos, os módulos:

Trees: Módulo com todas as árvores utilizadas pelo sistema.

Elements: Módulo com a hierarquia utilizada para a classe Element.

Formulation: Módulo com a hierarquia utilizada para a classe Formulation.

ControlNodes: Módulo com as classes: Controle responsável por todo o fluxo do framework, TakeFile responsável pela leitura e escrita nos arquivos e Node responsável pela manipulação dos nós.

MatVector: Módulo com as classes que manipulam os vetores e matrizes.

Materials and Sources: Módulo com as classes Materiais e Fontes.

Depois do diagrama de classes dividido em módulos, iniciamos a geração de todos os demais diagramas da UML. O primeiro a ser criado foi o diagrama de casos de uso principal, mostrado na figura 3.10.1. O diagrama funciona como um diagrama de contexto do sistema. Mostra a interação do framework com o meio externo. Os atores representam tudo o que pode estar sendo acessado no exterior. No caso do framework, esse diagrama é simples, mostrando somente a interação do sistema com o usuário, os arquivos de entrada e os arquivos de saída.

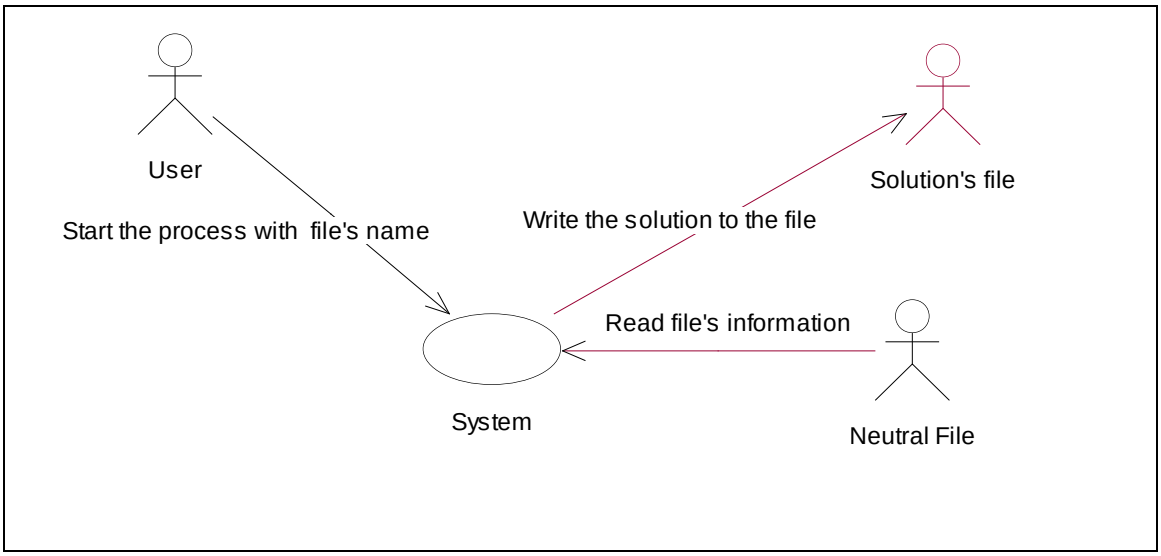

Figura 3.10.1: Principal diagrama de casos de uso do sistema

Os diagramas das figuras 3.10.2 e 3.10.3 são, respectivamente, os diagramas de seqüência e de colaboração. O primeiro tem o objetivo de informar uma seqüência de operações no tempo; o segundo, que pode ser gerado a partir do primeiro, mostra um cenário alternativo para as operações. Estes diagramas trazem informação redundante, por isto o diagrama de colaboração é mostrado somente neste primeiro caso. As figuras mostram no mais alto nível toda a seqüência de execuções gerenciadas pela classe Controle. Podemos observar que elas estão encadeados desde o usuário, informando os nomes dos arquivos que serão lidos, até as gravações dos resultados no arquivo de saída. Primeiro o usuário informa quais os arquivos que serão lidos, PutFile(). Com essa informação começa então o processamento. A classe Controle encadeia as operações no framework, faz a leitura dos dados dos Nós, Elementos, Materiais e Fontes. Depois de todos os dados armazenados, a classe Controle verifica com a função DecideProblem() se o framework está pronto para resolver o problema proposto. Caso esteja, realiza o cálculo por Elementos Finitos, soluciona o sistema matricial, guarda e escreve sua solução em um arquivo. A função WriteSolution(), última mostrada na figura 3.10.2, incorpora, além da gravação do arquivo de saída a solução do sistema matricial.

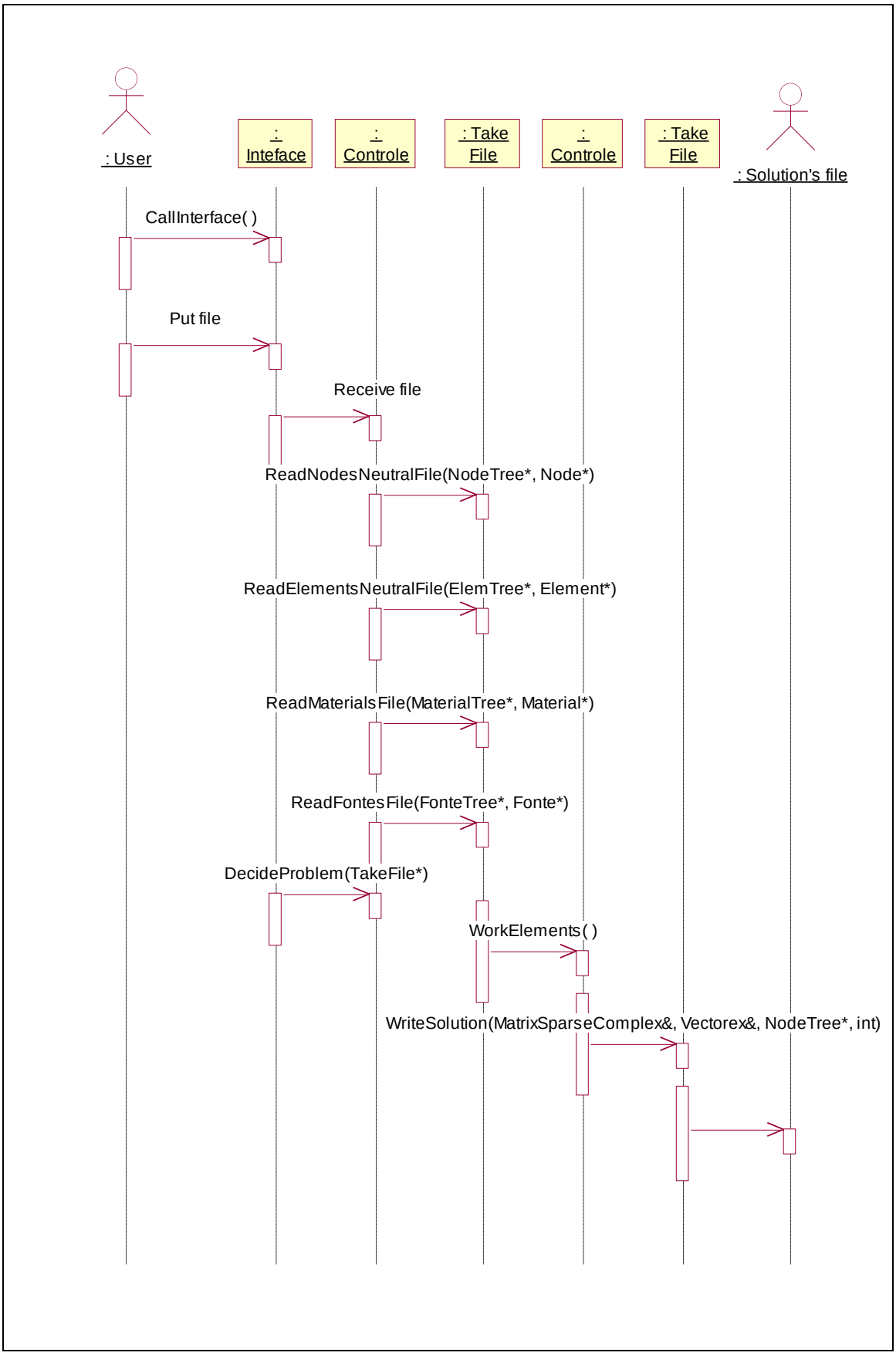

Figura 3.10.2: Diagrama de seqüências mais genérico do sistema.

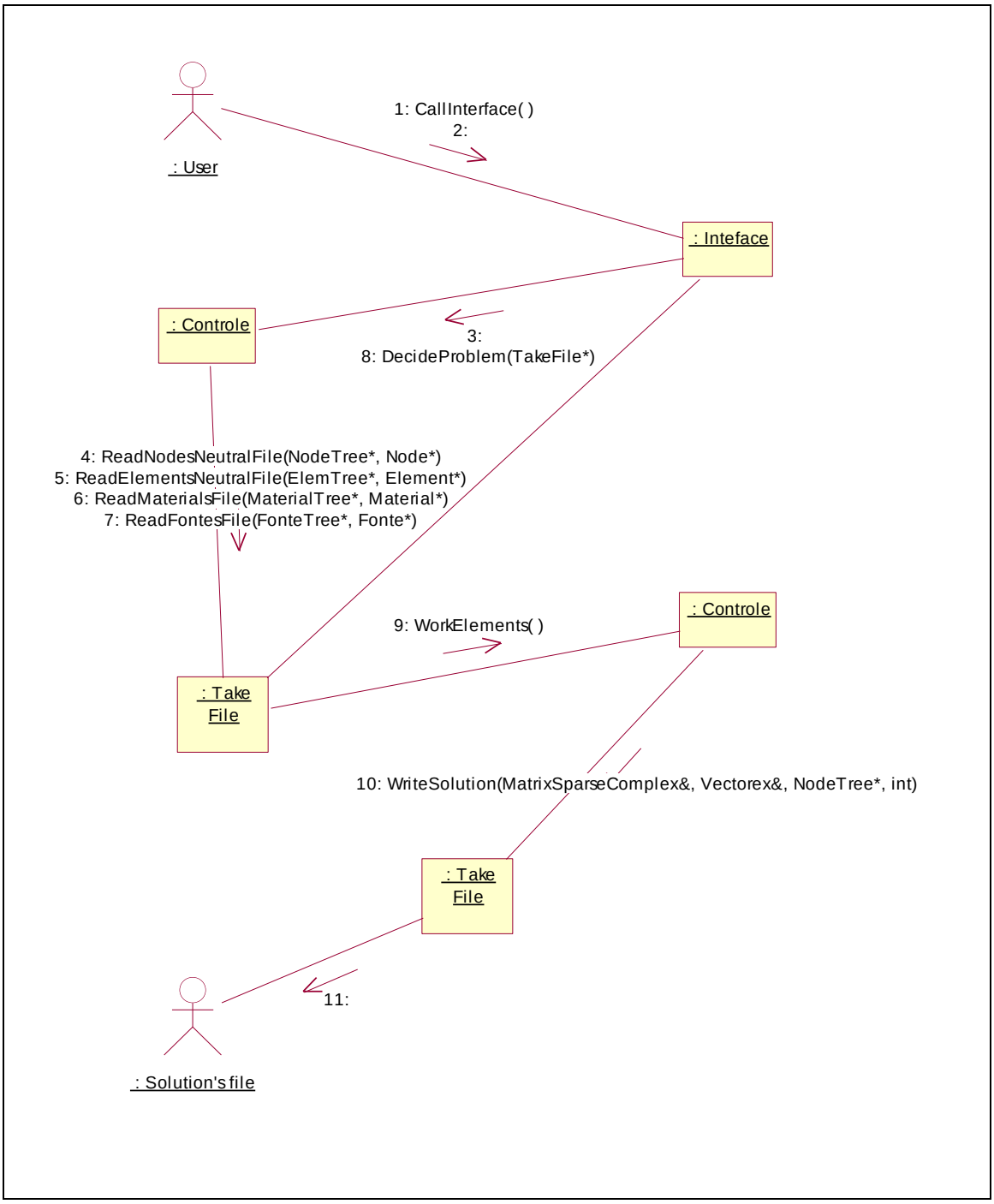

Figura 3.10.3: Diagrama de colaboração mais genérico do sistema.

O próximo diagrama, figura 3.10.4, já em um nível mais baixo, descreve apenas a classe Controle, manipulando juntamente com a classe TakeFile, as leituras dos arquivos de entrada.

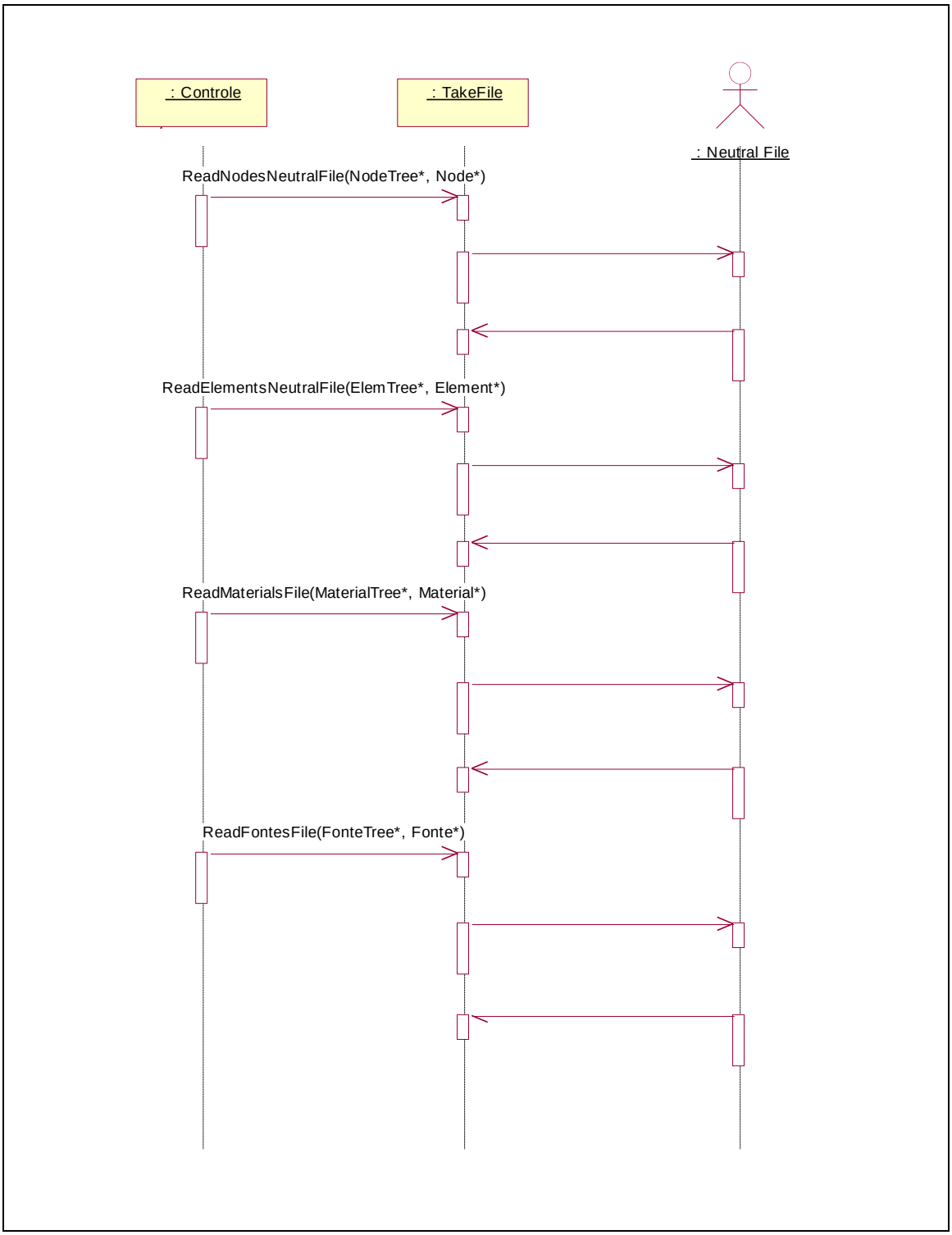

Figura 3.10.4: Diagrama de seqüência para a classe Controle, manipulando a classe TakeFile

Os diagrama mostrado na figura 3.10.5 exemplifica quais são as operações mais internas da classe Controle. Depois que todas as informações dos arquivos já foram lidas, a classe Controle inicia uma seqüência que é

praticamente interna à classe. Nesse caso, encontramos execuções simultaneamente recursivas, indicando que a classe realmente está executando operações suas. A UML possui outro diagrama, denominado diagrama de Estados, que mostra a interação dessas funções, conforme pode ser visto na figura 3.10.6.

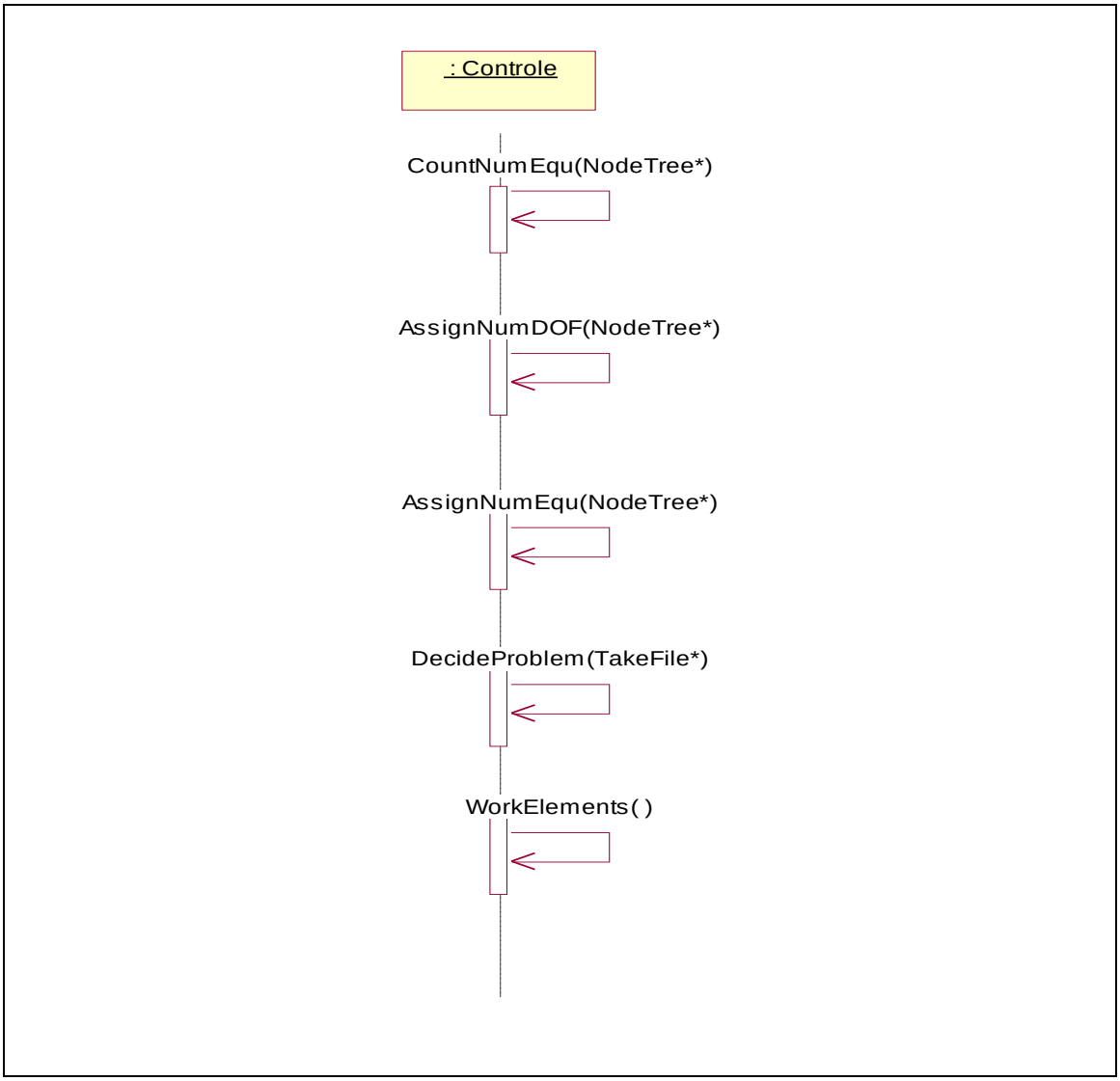

Figura 3.10.5: Diagrama de seqüência para as operações internas da classe Controle

O diagrama de Estados, mostrado na figura 3.10.6, permite uma maior visualização dos estados internos de uma classe. Nele é fundamental observar a importância da função DecideProblem() que verifica se o framework está preparado para trabalhar os dados recebidos nos arquivos de entrada. Essa

função permite, ou não, a continuação dos processos, dependendo dos dados recebidos.

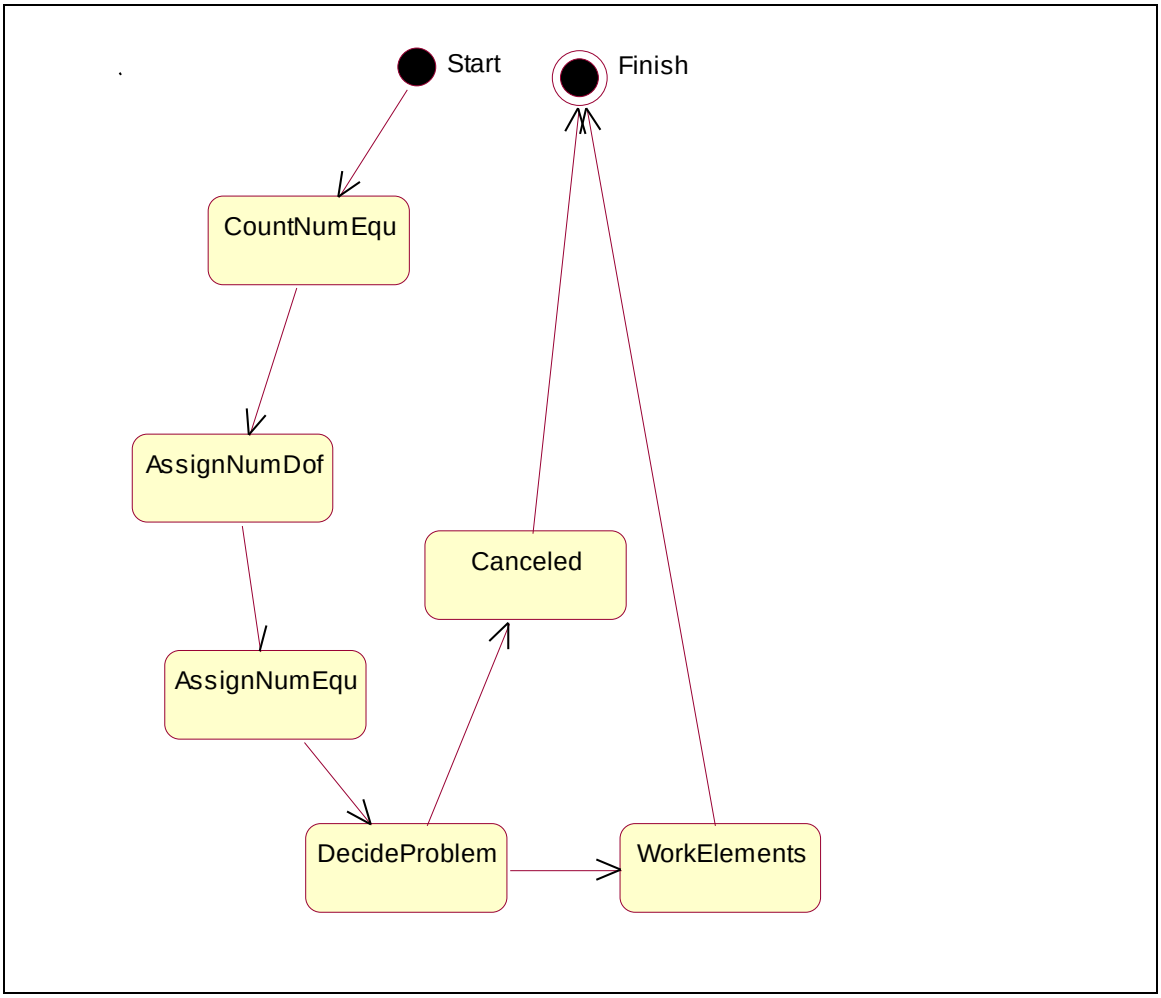

Figura 3.10.6: Diagrama de Estados para as operações internas da classe Controle

O próximo diagrama de Seqüência, figura 3.10.7, mostra as operações para a última função da figura 3.10.6: WorkElements() . Ele mostra a seqüência de operações de todos os elementos, independente de suas características, até a gravação dos resultados finais. Inicialmente é acionada a função ComputeLocalMatriz() que gera a matriz de contribuições para o elemento que está sendo trabalhado; em seguida função SetLocationArray() monta a matriz LM, matriz que possui os nós com suas numerações ou a indicação que possuem condição de contorno. Depois da matriz de contribuições do elemento calculada e a matriz LM montada, através da função AddElem(), essas contribuições são compostas na matriz principal. Os próximos dois passos

dizem respeito ao vetor de contribuições. Primeiramente a função ComputeLocalVetor() verifica se aquele elemento, com seus respectivos nós, geram contribuição, calculando-as se necessário. Depois a função AddElemFonte() faz a composição dessas contribuições no vetor principal. A função SetBoundaryConditions() resgata as condições de contorno dos nós do elemento para que então a função AddElem() possa, fazendo os cálculos necessários, compor o resultado dessas contribuições no vetor global de contribuições. Depois que esse processo é feito para todos os elementos então é acionada a função WriteSolution(), que soluciona esse sistema e grava os valores encontrados para cada nó no arquivo de saída.

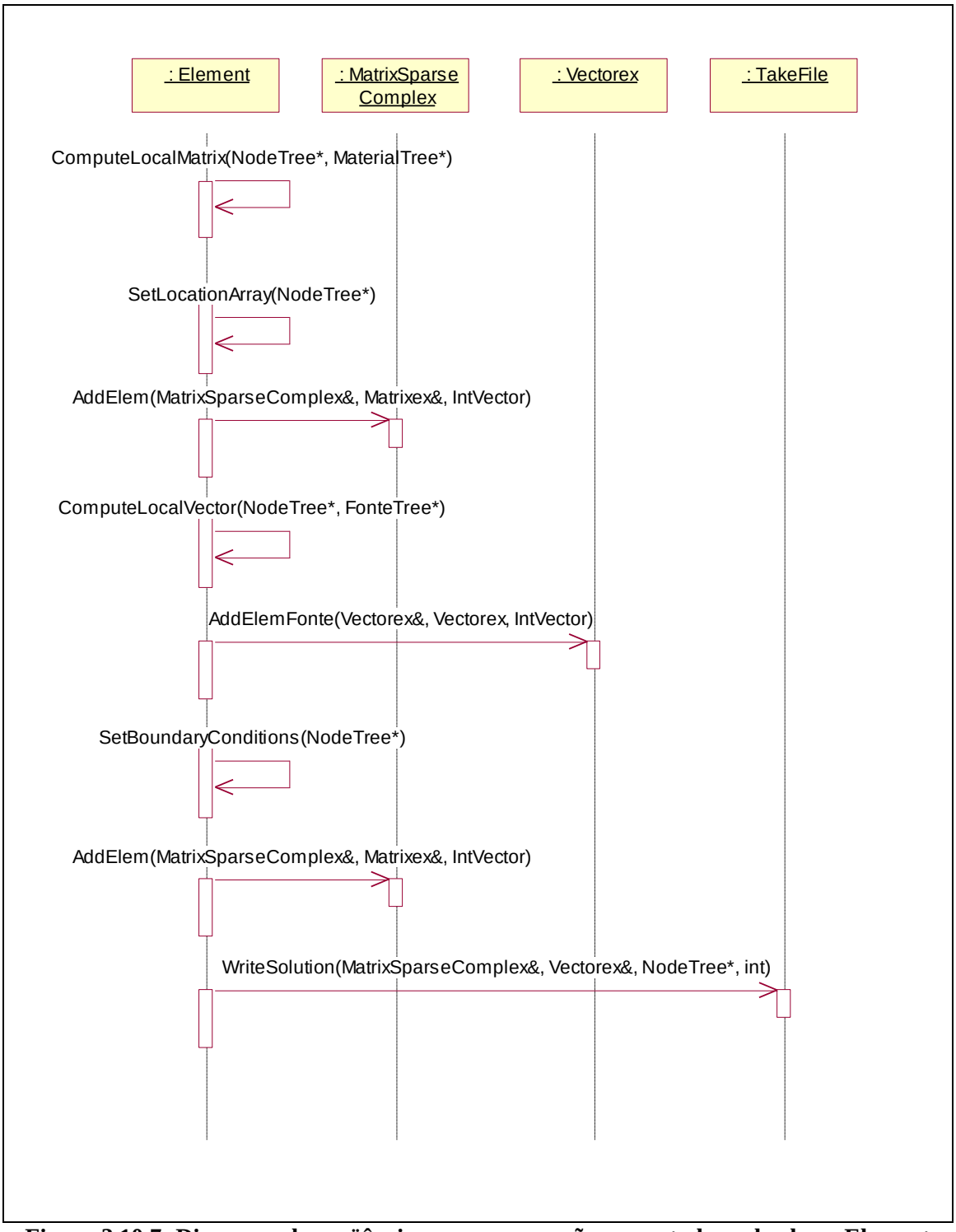

Figura 3.10.7: Diagrama de seqüência para as operações executadas pela classe Element

O próximo diagrama de Estados, figura 3.10.9, mostra as funções ComputeLocalMatrix() e ComputeLocalVector(). Essas funções têm suas operações conhecidas pela comunidade científica da área de elementos finitos, mas aqui, para se obter maior generalidade, nós utilizamos um duplo

despacho. Podemos observar que o despacho está presente nas duas funções. Essas funções ficaram executando apenas o que é comum para cada tipo de elemento (determinar quais são suas funções de forma, pontos de integração, efetuar integração numérica, calcular matriz do jacobiano, etc). O que é diferente, na formulação de cada tipo de problema, é resolvido pela classe Formulation. No estado de despacho, Dispatching, todas as informações necessárias são despachadas para a classe Formulation que , resolvendo o problema, retorna o controle, com os devidos parâmetros, para a função terminar o seu processamento. Assim, Element fica sabendo que formulação está sendo empregada e consegue saber o que será integrado por ele para compor a contribuição.

Para explicarmos o despacho duplo é interessante fazermos uma breve explanação sobre o despacho simples, que nada mais é que o polimorfismo [Boo94] dependente somente de uma classe: quando temos uma classe complexa como a classe Element, que se especializa em vários tipos de elementos, e ainda, não sabemos, antes do tempo de execução, quais os tipos que serão utilizados, precisamos de tomar algumas resoluções durante o processamento. Essas resoluções em tempo de processamento é que nos dão a idéia de muitas formas, o polimorfismo. Quando o processamento encontra um elemento e se chama uma função genérica sobre ele (como, por exemplo, a função de cálculo de sua matriz de contribuições) a existência do polimorfismo permite a ligação do elemento à função correta de cálculo.

Suponhamos, agora, uma execução de função dependente de duas classes: no caso as classes Element e Formulation. Fazemos, então, um primeiro despacho de função que executa uma função polimórfica de um tipo de elemento (o polimorfismo age pela primeira vez). Chama-se, então, uma função polimórfica de Formulation e tem-se como resultado, um polimorfismo duplo, onde o que vai ser calculado depende do tipo de elemento e do tipo de formulação que estão sendo empregados. Seguindo esse raciocínio, podem ser gerados despachos duplos, triplos, etc, dependendo da necessidade do problema.

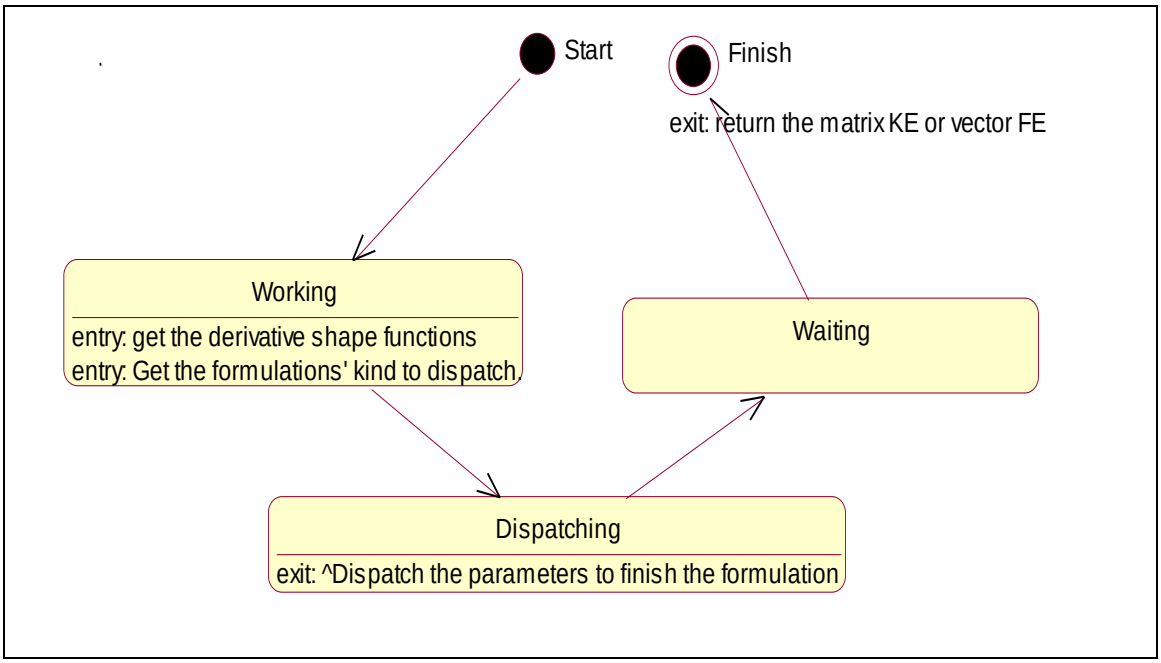

Figura 3.10.9: Diagrama de estados mostrando o duplo despacho

3.11 Exemplificação das funcionalidades do framework

É muito importante mostrarmos a facilidade para implementação de novas características no framework. Suponhamos que o framework fosse agregar também a resolução de problemas em três dimensões. O fluxo de controle permanece o mesmo, o processo de leitura já está pronto, teremos que agrupar à hierarquia da classe Element os elementos de terceira dimensão, observando que todos os métodos comuns a todas as dimensões, presentes no topo da hierarquia, serão automaticamente herdados. Logicamente, teremos ainda que agregar à classe Formulation as formulações necessárias para as soluções dos problemas propostos. Os métodos específicos dos elementos e das formulações podem não ser muito simples mas apenas eles terão que ser programados, a tabela 1 mostra uma síntese da parte herdada e da parte que será desenvolvida:

Parte Herdada	Parte que será desenvolvida
Processo de Leitura	
Fluxo de controle	**Agregação do tipo de problema**
Funções definidas em todos os primeiros níveis	**Funções exclusivas para resolução em 3D:** **Integradas a classe Element e a Classe Formulation.**
Gravação dos resultados	

Tabela 1: Partes herdadas e a serem desenvolvidas em uma nova implementação

A figura 3.11.1 mostra o início da implementação na Classe Element dos elementos de três dimensões:

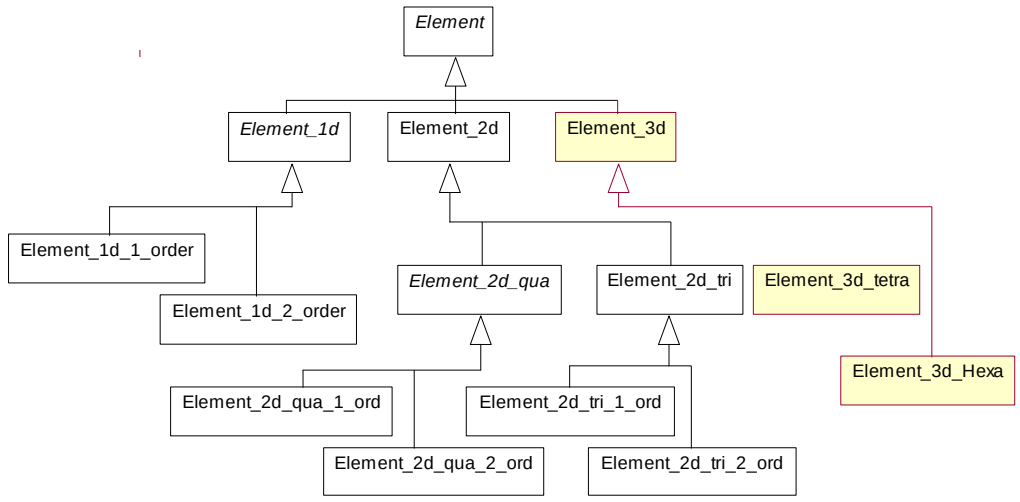

Figura 3.11.1: Diagrama de Classes mostrando o início da implementação dos elemento 3D

A figura 3.11.2 mostra o início da implementação na classe Formulation das formulações referentes aos problemas de três dimensões:

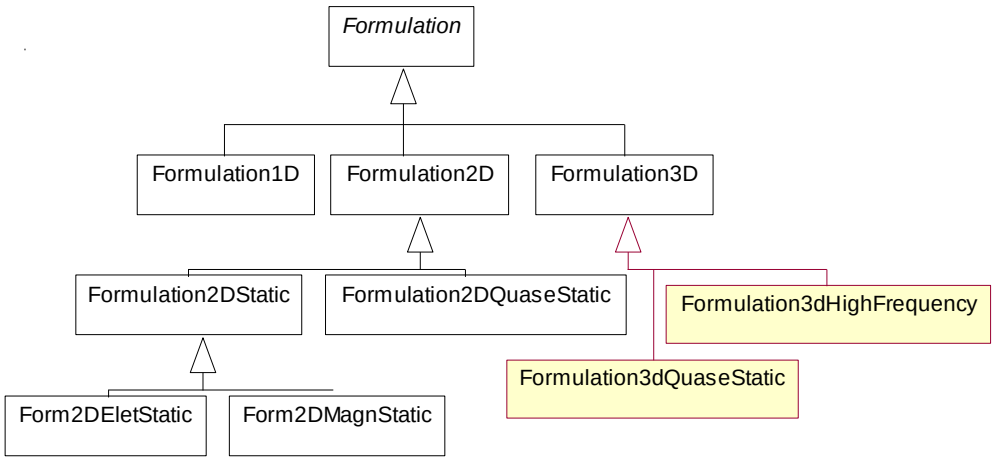

Figura 3.11.2: Diagrama de Classes mostrando o início da implementação das formulações para os elemento 3D

Esse foi apenas um exemplo da viabilidade e facilidade de uso dos frameworks. É muito importante não esquecermos também as facilidades encontradas durante a geração dos minissistemas, onde estávamos a todo momento evoluindo o framework.

4 - Conclusão

4.1 Considerações iniciais

O objetivo desse trabalho, a criação de um framework, utilizando o método de Elementos Finitos, para solução de problemas Estáticos e Quase-Estáticos, para uma e duas dimensões, foi atingido. Este último capítulo apresenta as nossas principais conclusões sobre o projeto, depois da sua finalização. Apresenta, ainda, propostas para trabalhos futuros.

É extremamente importante lembrar que o principal objetivo dos frameworks é gerar a facilidade de reutilização de código e conseqüente evolução do sistema. Dessa forma, as propostas para trabalhos futuros devem ser muito bem aproveitadas. Elas darão vida ao framework construído.

4.2 Conclusão geral do trabalho

Esse trabalho foi apresentando resultados durante todo o seu desenvolvimento. A exposição desses resultados será temporal e na ordem em que foram sendo descobertos.

O desenvolvimento de frameworks é realmente mais difícil do que o desenvolvimento de softwares comuns. O conhecimento do problema e a capacidade de generalização têm que ser muito grandes e isso só se consegue com o tempo e muito trabalho.

Se fôssemos comparar a dificuldade para geração dos frameworks com a dificuldade para aproveitamento dos seus benefícios, poderíamos dizer que são, neste trabalho, inversamente proporcionais. A partir do momento em que o framework começou a tomar forma, o desenvolvimento dos minissistemas ficou cada vez mais simples, pois o código ia sendo herdado, tendo assim muito menos para ser desenvolvido.

A experiência é muito importante para diagnosticar que o método de Elementos Finitos realmente possui características de frameworks e esse fator contribuiu muito para o sucesso do trabalho. É importante ressaltar a estrutura genérica que se obteve, com separação clara entre o que é característica de elemento e de formulação, o que facilita a introdução de novos elementos para formulações já existentes ou de novas formulações para elementos já existentes sem alteração dos mesmos.

O framework gerado provou como previsto na teoria [Wil93] ser um produto de fácil manutenção e testes. Como possui um fluxo fixo pudemos observar que esse fluxo foi testado intensamente durante o desenvolvimento dos minissistemas. Em caso de manutenção, existem funções que, por estarem nas especializações superiores das classes e serem utilizadas por vários problemas, estão acima de qualquer suspeita. Isso facilita a manutenção e é objetivo dos frameworks, o que também foi alcançado aqui.

Em resumo: a utilização de frameworks é extremamente interessante, trazendo eficiência nas evoluções, manutenções e testes dos sistema.

4.3 Sugestões para futuros trabalhos

O framework desenvolvido não explora problemas de alta freqüência, e nenhum tipo de problema em três dimensões. Deixamos, então, como sugestão, o desenvolvimento desses problemas. Esses trabalhos deixarão o produto bem abrangente.

O framework utiliza algumas caixas pretas, pacotes que utilizamos sem nos preocupar com o seu conteúdo. Esses pacotes não estão construídos totalmente orientados a objetos e seria um trabalho interessante modificá-los para seguir este modelo.

5 - Referências bibliográficas

[Bar94] BARTON J, NACKMAN L.R. Scientific and engineering C++: an introduction with advanced techniques and examples. Addison Wesley Longman,1994. 671 p.

[Boo94] BOOCH, G. Object-oriented analysis and design. The benjamin/cummings publishing company, may 1994.

[Bom98] BOMME P, EYHERAMENDY D, VERNIER L, ZIMMERMANN T. Aspects of an object-oriented finite element environment. Computers & Structures 68: (1-3) pp.1-16 jul-aug 1998.

[Cot95] COTTER S, POTEL M. Inside taligent technology. Addison-Wesley, 1995.

[Deu89] DEUTSCH L.P. Design reuse and frameworks in the samlltalk-80 system. in T.J. Biggerstaff and A.J. perlis, editors, Software Reusability, v. II, ACM Press, 1989.

[Fir94] FIRESMITH D.G. Frameworks: the golden path to object Nirvana. Journal of object_oriented programming, v. 7 No. 8, 1994.

[Fow97] FOWLER M. Applying the standard object modeling language. Addison Wesley Longman, 1997.

[Hug87] HUGHES T. The finite element method. Prentice-Hall, 1987.

[Jim93] JIM J. The Finite Element Method in Electromagnetics. John Wiley & Sons, Inc, 1993.

[Joh91] JOHNSON R.E. Reusing object-oriented design. University of Illinois. Technicall Report UIUCDCS 91-1696, 1991.[26]

[Joh93] JOHNSON R.E. How to design frameworks, tutorial notes. In 8th conference on Object-Oriented Programming Systems, Languages and Applications, Washington, USA, 1993.

[Jon98] JONHSON R.E, FOOTE B. Designing reusable classes. Journal of object-oriented programming, v. 1, No. 2, June 1998.

[Kal95] KARLSSON E-A. Software reuse – a holistic approach. John Wiley & Sons, 1995.

[Koe97] KOENIG A, MOO B. Ruminations on C++. AT&T, 1997.

[Lom97] LOMÔNACO A. G. Métodos computacionais aplicados à resolução de problemas de alta freqüência em eletromagnetismo, Dissertação de Mestrado, PPGEE / UFMG, 1997.

[Mai00] MAI W, HENNEBERG G. Object-oriented design of finite element calculations with respect to coupled problems. IEEE Transactions on Magnetics in press, 2000.

[Mac98] MACKIE R.I. An object-oriented approach to fully interactive finite element software. Advances in engineering software v. 29, No. 2, pp. 139-149, 1998.

[Mac99] MACKIE R.I. Object-oriented finite element programming the importance of data modelling. Advances in Engineering Software v. 30, pp. 775-782, 1999.

[Mag00] MAGALHÃES, A. L. C. C. Estudo, projeto e implementacão de um modelador de sólidos voltado para aplicações em eletromagnetismo. Belo Horizonte: PPGEE-UFMG, 2000. (Tese, Doutorado em Engenharia Elétrica), Programa de Pós-Graduação em Engenharia Elétrica - UFMG, 2000.

[Mat96] MATTSON M. Object-oriented frameworks – A survey of methodological issues. Lutedx/(tecs-3066) 1-130/(1996).

[Oht93] OHTSUBO H, KAWAMURA Y, KUBOTA A. Development of the object-oriented finite-element Modeling System. Modify eng. comput 9: (4) 187-197 1993.

[Qua98] QUATRANI T. Visual modeling with Rational Rose and UML. Addison Wesley Longman, 1998.

[Rus90] RUSSO V.F. An object-oriented operating system. PhD. Thesis, University of Illinois at Urbana-Champaign, October 1990.

[Sil94] SILVA EJ, MESQUITA R, SALDANHA RR, PALMEIRA PFM. An Object-oriented Finite-element Program for Electromagnetic-field Computation. IEEE Transactions on Magnetics 30: (5) 3618-3621, part 2 Sep 1994.

[Str98] STROUSTRUP B. The C++ Programming Language. Addison Wesley Longman, 1998. 911 p.

[Wil93] WILSON D.A, WILSON S.D. Writing frameworks-capture your expertise about a problem domain, tutorial notes. In the 8th conference on object-oriented programming systems, Language and Applications, Washington, 1993.

[Wei89] WEINAND A, GAMMA E, MARTY R. Design and implementation of ET++, A seam less object-oriented application framework. Structured Programming, v. 10, No. 2, July 1989.

[YU93] YU G, ADELI H. Object-oriented Finite-element Analysis using EER Model. J struct eng-asce 119: (9) 2763-2781 Sep 1993.

[Zim96] ZIMMERMANN T, EYHERAMENDY D. Object oriented finite element programming: An interactive environment for symbolic derivations, application to an initial boundary value problem. Advances in Engineering Software 27: (1-2) 3-10 oct-nov 1996.