UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Modelagem do Projeto do Software FDTD Studio

Arthur Wesley Oliveira Leite

Brasília, dezembro de 2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Modelagem do Projeto do Software FDTD Studio

Arthur Wesley Oliveira Leite

MONOGRAFIA DE PROJETO FINAL SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRICISTA.

APROVADA POR:

Prof. Antonio José Martins Soares, Doutor (UNICAMP)
(Orientador)

Prof. Leonardo R.A.X. de Menezes, Doutor (University Of Victoria)
(Examinador Interno)

Prof. Renato Proença Picanço, Mestre (UnB)
(Examinador Externo)

RESUMO

O presente trabalho tem como objetivo a modelagem e reestruturação do software FDTD Studio, programa desenvolvido para a solução das equações de Maxwell por meio do método das diferenças finitas no domínio do tempo. Pretende-se, ainda, elaborar uma documentação que possa servir de base para um estudo posterior e aperfeiçoamento do programa. São apresentados diversos conceitos relacionados à Engenharia de Software, que serão utilizados como ferramentas e diretrizes para a estruturação do novo sistema. É feito um estudo do programa original mostrando suas funcionalidades básicas e como ocorre o relacionamento entre seus principais elementos. Por fim, é realizado o projeto do novo software a partir de uma organização lógica de seus componentes. São criados componentes fundamentais com o objetivo de isolar complexidades e fornecer uma maior modularidade ao programa. Incluem-se diversos diagramas que ilustram como o software será organizado e como serão implementadas várias de suas funcionalidades.

ABSTRACT

The present work aims at modeling and restructuring of the FDTD Studio software program developed for the solution of Maxwell's equations using the finite difference method in time domain. The aim is to also prepare documentation that can serve as a basis for further study and improvement of the program. We present several concepts related to software engineering, which will be used as tools and guidelines for establishing the new system. It is shown the original program showing its basic features and as is the relationship between its main components. Finally, the project of the new software is done from a logical organization of its components. Fundamental components are created in order to isolate complexities and provide greater modularity to the program. Are presented several diagrams that illustrate how the software is organized and how will be implemented several of its features.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

DLL	*Dynamic Link Library*
FDTD	*Finite Difference Time Domain*
GoF	*Gang of Four*
GRASP	*General Responsibility Assignment Software Patters*
IDE	*Integrated Development Environment*
PML	*Perfect Matched Layer*
RAD	*Rapid Application Development*
RUP	*Rational Unified Process*
UML	*Unified Modeling Language*

1. INTRODUÇÃO

A solução analítica para as equações diferenciais parciais de Maxwell é possível apenas em problemas ideais, didáticos. Em uma aplicação mais próxima da realidade, as soluções manuais são impraticáveis, sendo comum a utilização de métodos numéricos computacionais para a obtenção de soluções aproximadas para os problemas. O uso desses métodos se tornou recorrente, principalmente com o barateamento dos computadores e o desenvolvimento intensivo da capacidade de processamento.

Os métodos existentes para a resolução das equações de Maxwell baseiam-se na solução no domínio do tempo ou no domínio da freqüência. Os métodos baseados na solução no domínio da freqüência apresentam algumas limitações, em especial na análise de elementos feitos de material dielétrico [1]. Entre os métodos que se baseiam na solução no domínio do tempo, se encontra o Método das Diferenças Finitas no Domínio do Tempo (*Finite Difference Time Domain* - FDTD). Ele serviu de base para a elaboração do software FDTD Studio, utilizado para análise e projeto de antenas.

O FDTD Studio é bem completo e reúne inúmeras funcionalidades necessárias para a análise e modelagem de diversas estruturas. Foi desenvolvido utilizando o paradigma da orientação a objetos de forma que sua estrutura é organizada em classes que se relacionam para o bom funcionamento do sistema.

Para a sua elaboração, foram utilizados diversas ferramentas e softwares auxiliares tais como compiladores e ambientes de desenvolvimento integrado. Em especial, utilizou-se o *Borland Developer Studio* como IDE principal. Isto proporcionou o acesso a bibliotecas exclusivas, que, apesar de serem bastante úteis e práticas, não são de domínio livre, tornando a manutenção e desenvolvimento do software restrito àqueles que possuem acesso a esse ambiente. Procurou-se, neste projeto, utilizar, ao máximo, ferramentas de desenvolvimento livres. Como IDE principal, por exemplo, foi utilizado o *Code Blocks*, uma IDE livre e bastante completa. Alguns elementos restritos à biblioteca da *Borland* foram substituídos por outros equivalentes da *C++ Standart Library*. Esses elementos não estão mostrados em detalhes neste trabalho, mas podem ser facilmente identificados no código fonte.

Na elaboração do novo software, foi aproveitada toda a estrutura básica do programa original. Primeiro, fez-se um isolamento das classes principais de forma a poder acessá-las diretamente pelo código fonte, sem o auxílio da interface. A partir disso, foi

possível construir todos os outros elementos que irão se relacionar com essa estrutura para que os modelos possam ser simulados eficientemente.

Os componentes foram, inicialmente, organizados em pacotes funcionais. Isso trouxe um melhor entendimento de como as estruturas do software irão se comunicar durante a sua execução. Foi mostrada como essas estruturas se relacionam e quais elementos a compõe. Diversos diagramas foram utilizados na modelagem de forma a auxiliá-la e, ao mesmo tempo, fornecer uma documentação útil que possa ser usada, posteriormente, para o estudo e expansão do software.

No capítulo 2, são mostrados diversos recursos relacionados à Engenharia de Software. É apresentada a Linguagem de Modelagem Unificada, utilizada para a modelagem de sistemas orientados a objetos. É descrito o processo de desenvolvimento de software denominado desenvolvimento iterativo e incremental, consagrado na elaboração de sistemas grandes e robustos. Também são mostrados diversos padrões de projetos existentes, que caracterizam princípios de programação amplamente testados e reconhecidos. Por fim, é descrita a linguagem de programação utilizada, *C++*, e o ambiente de desenvolvimento integrado, *Code Blocks.*

No capítulo 3, é feita uma análise do software FDTD Studio. Primeiramente, é mostrado como ocorre o funcionamento básico do programa. É tecido, então, uma análise de suas principais classes, mostrando seus aspectos fundamentais, o funcionamento de suas principais funções e relacionamento entre elas. Para isso foram utilizados inúmeros diagramas, além da análise de trechos de código específicos.

No capítulo 4, é descrito o projeto do novo programa. Primeiramente, são mostradas as suas principais funcionalidades por meio do diagrama de casos de uso do software. É feita, então, a organização dos componentes do sistema através de pacotes lógico funcionais. Tais pacotes abrangem todos os elementos do software, inclusive aqueles que venham a ser criados em um estudo posterior. É descrito, então, os diversos componentes criados neste trabalho e o relacionamento entre eles usando-se diagramas de sequência. No final, é mostrada a tela principal do novo software além de recomendações para um aperfeiçoamento posterior das classes do programa.

2. ENGENHARIA DE SOFTWARE

Engenharia de Software é uma área da computação voltada para a manutenção e desenvolvimento de sistemas computacionais. Ela utiliza um conjunto de métodos, técnicas e ferramentas para analisar, projetar e gerenciar o desenvolvimento e manutenção de softwares.

No presente capítulo, serão mostrados diversos recursos relacionados a essa área e que serão utilizados no desenvolvimento da nova versão do FDTD Studio.

2.1. A Linguagem de Modelagem Unificada - UML

UML é a linguagem padrão de modelagem adotada internacionalmente pela indústria de Engenharia de Software. Com ela é possível visualizar por meio de diagramas padronizados o trabalho desenvolvido pelos programadores. Ela é a ferramenta mais adequada para que se possa especificar, documentar e elaborar o projeto de softwares que utilizam o paradigma de orientação a objetos. Vale ressaltar que a UML não é uma linguagem de programação, mas sim de modelagem. Ela ajuda os engenheiros a definir requisitos, comportamento e estrutura lógica dos programas.

A UML é composta por 14 diferentes tipos de diagramas que possuem o objetivo de fornecer múltiplas visões do sistema a ser modelado. Procura-se dessa forma uma modelagem completa do sistema em que os diferentes diagramas se complementam. Com os diagramas é possível analisar o sistema a partir de diferentes níveis, podendo focalizar, por exemplo, a sua estrutura, algum processo específico ou, até mesmo, componentes físicos necessários. As utilizações de diferentes diagramas auxiliam inclusive na descoberta de falhas, evitando erros futuramente.

No entanto, apesar da grande variedade de diagramas existentes na linguagem, não é obrigatório a utilização de todos eles na modelagem do sistema visto que, dependendo do projeto a ser elaborado, alguns deles são desnecessários. A seguir serão descritos os diagramas utilizados na modelagem do FDTD Studio.

2.1.1. Diagramas de Casos de Uso

O Diagrama de Casos de Uso é utilizado no início da modelagem, durante a análise de requisitos do sistema, documentando e especificando principais funções e serviços que serão oferecidos ao usuário pelo software. Ele tenta apresentar o sistema a partir de uma perspectiva do usuário e é o mais abstrato e informal de todos. A partir dele, é possível ter uma visão externa geral do software sem se preocupar em como as funções serão implementadas. Ele é consultado durante todo o processo de desenvolvimento do sistema.

Os Diagramas de Casos de Uso são representados utilizando dois componentes: Atores e Casos de Uso.

Os Atores equivalem aos usuários que interagem com o software. A notação utilizada para designá-los está mostrada nos exemplos da figura 2.1. Os Casos de Uso representam funções ou tarefas fornecidas pelo software ao usuário. Eles são utilizados para documentar as funções fornecidas pelo sistema e são representados por elipses contendo um texto no seu interior que representa a função especificada. A figura 2.2 mostra um exemplo de Caso de Uso. Os Casos de Uso costumam ser documentados de forma a estabelecer quais os atores que o executam, em quais condições são executados, quais são as pré e pós condições necessárias, quais etapas devem ser executadas e quais restrições são impostas.

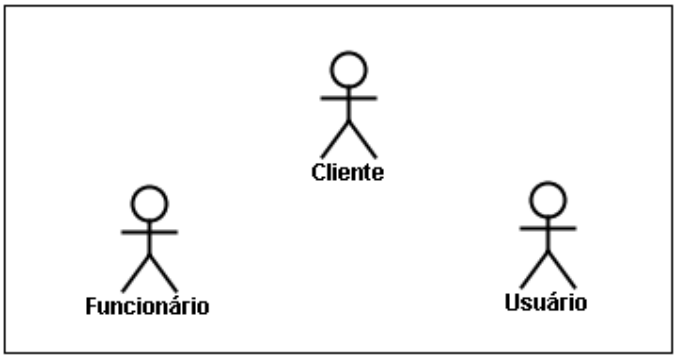

Figura 2.1. Exemplo de Atores

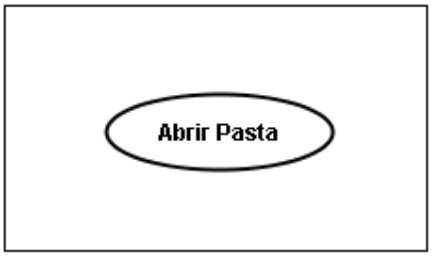

Figura 2.2. Exemplo de Caso de Uso

Não existe um formato específico para se documentar um caso de uso, devido à flexibilidade da linguagem. A referência [2] fornece o modelo de documentação mostrado na Tabela 2.1.

Tabela 2.1. Modelo de Documentação de Casos de Uso

Nome do Caso de Uso	**Exibir Resultados**
Caso de Uso Geral	
Ator Principal	
Atores Secundários	
Resumo	
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
Restrições/Validações	

O parâmetro Caso de Uso Geral deve ser especificado quando o Caso de Uso descrito for um caso especial de um Caso de Uso mais abrangente, tendo com este uma associação do tipo Generalização como será mostrado. O parâmetro Ator Principal refere-se ao ator que mais interage com o caso de uso ou que está mais interessado nos resultados fornecidos por ele. Já os Atores Secundários são aqueles que não estão tão interessados nos resultados fornecidos pelo Caso de Uso.

É fornecido também um breve resumo das funcionalidades do Caso de Uso e as pré e pós condições necessárias, ou seja, em quais circunstâncias o Caso de Uso pode ser executado e quais são as ações ou funções que devem existir após a sua chamada.

Os campos Ações do Ator e Ações do Sistema devem fornecer uma sequência lógica de atividades realizadas pelo ator principal e pelo sistema. Se desejado, pode-se adicionar novas ações referentes a algum cenário alternativo, se o Caso de Uso não for executado na sua forma mais usual. O campo Restrições/Validações deve conter possíveis informações adicionais para tornar a processo mais consistente. Os Diagramas de Casos de Uso também são caracterizados pela existência de Associações entre seus elementos. As Associações mais comuns ocorrem entre os Atores e os Casos de Uso e indicam que o Ator em questão executa o Caso de Uso enviando informações, recebendo resultados ou ambos.

As Associações são representadas por retas que podem possuir uma seta em alguma de suas extremidades representando a navegabilidade das informações. A figura 2. 3 ilustra

uma associação entre Ator e Caso de Uso. Como pode-se observar, o Ator Usuário solicita de alguma forma o serviço Pesquisar Nome ao sistema.

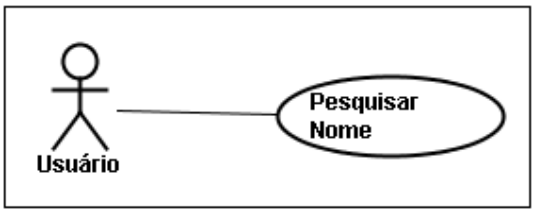

Figura 2.3. Associação entre Ator e Caso de Uso

As Associações também podem ser do tipo Generalização, Inclusão ou Extensão. A Associação do tipo Generalização ocorre quando existe dois ou mais Casos de Uso com características parecidas. Assim, é criado um Caso de Uso Geral que possui as características comuns entre eles e é criada uma associação do tipo Generalização entre os Casos de Usos específicos e o Caso de Uso Geral. Esse tipo de relacionamento é importante, pois se economiza documentação, visto que as características semelhantes são descritas apenas uma vez já que são herdadas pelos Casos de Uso específicos. Herdam-se também os relacionamentos do Caso de Uso Geral com Atores ou outros Casos de Uso. A figura 2.4 ilustra uma Associação do Tipo Generalização.

Como mostrado na figura 2. 4, os Casos de Uso Plotar Gráfico de Barras e Plotar Gráfico de Pontos são Casos de Uso Específicos do Caso de Uso Plotar Gráfico. Portanto, na documentação desses Casos de Uso basta descrever as características específicas que não possuem na documentação mais geral do Caso de Uso Plotar Gráfico.

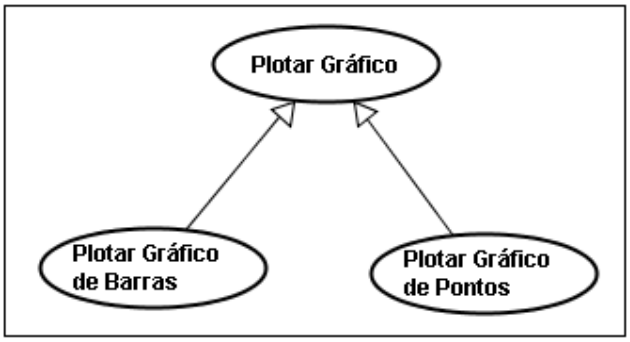

Figura 2.4. Associação do Tipo Generalização

A Associação do tipo Inclusão indica que a execução de determinado Caso de Uso obriga a execução de outro. Ela ocorre, em geral, quando existe uma função comum a mais

de um Caso de Uso. Essa Associação é descrita por uma reta tracejada com uma seta indicando o Caso de Uso incluído. A figura 2. 5 ilustra uma Associação do tipo Inclusão. Como pode ser observada, a solicitação dos Casos de Uso Fechar Janela ou Abrir Arquivo obriga a execução do Caso de Uso Salvar Dados.

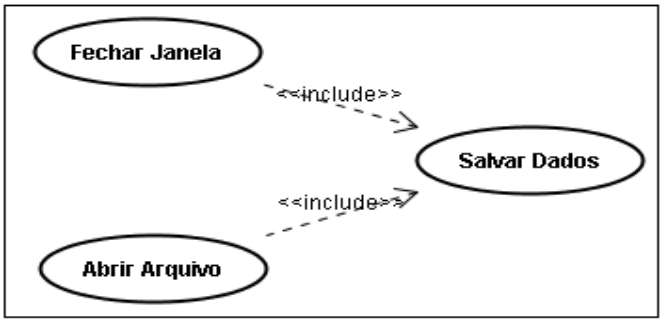

Figura 2.5. Associação do tipo Inclusão

A Associação do tipo Extensão indica que determinado Caso de Uso apenas pode ser executado em situações especificas, é preciso a execução de outro Caso de Uso para que ele possa ser chamado. Ela é representada por uma reta tracejada com uma seta indicando o Caso de Uso que solicita o Caso de Uso estendido. A figura 2.6 ilustra uma Associação do tipo Extensão. Como mostrado nessa figura, para o Caso de Uso Exibir Resultado ser chamado, é necessário a execução prévia do Caso de Uso Processar Dados.

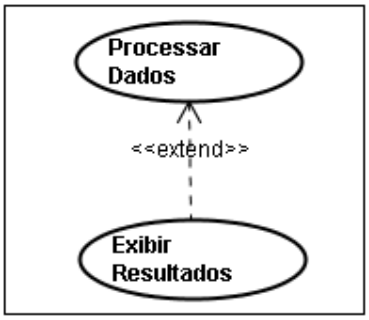

Figura 2.6. Associação do tipo Extensão

2.1.2. Diagrama de Classes

O Diagrama de Classes é o mais importante e o mais utilizado diagrama da UML. Com ele, é definida a estrutura das classes do sistema juntamente com os atributos e métodos respectivos. Apresenta também o relacionamento existente entre as classes e como ocorre a troca de informação entre elas. O Diagrama de Classes fornece uma visão

estática do software e serve de base para construção da maioria dos outros diagramas. A figura 2.7 mostra um exemplo de Classe utilizando a notação da UML. Pode-se ver que a notação para as classes na UML possui três divisões. A primeira refere-se ao nome da classe, a segunda aos atributos da classe e a terceira aos métodos da classe. Os sinais de mais e menos se referem, respectivamente, a visibilidade privada ou pública, que são características da programação orientada a objetos. Caso a visibilidade fosse protegida, seria utilizado o símbolo sustenido (#).

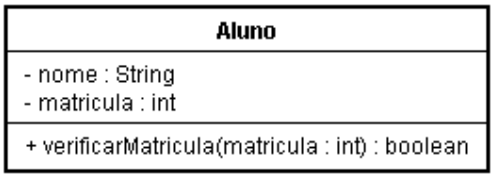

Figura 2.7. Classe

O Diagrama de Classes pode possuir diversos tipos de relacionamentos entre os componentes. Alguns dos principais tipos de relacionamentos e que serão utilizados no presente trabalho são Associação, Composição e Especialização/Generalização.

A associação indica que as instâncias de uma classe estão de alguma maneira relacionadas às instâncias das outras classes envolvidas no relacionamento. Ela é representada por uma reta ligando as classes envolvidas, podendo ou não possuir uma seta em uma das extremidades para simbolizar o sentido de navegação. É possível também a existência de um nome que caracterize associação. A figura 2.8 mostra um exemplo de Associação entre duas classes. Observa-se que existe um relacionamento entre a classe Aluno e a classe Professor. Os símbolos 0..* e 1..7 representam a multiplicidade que neste caso indica que cada instância da classe Aluno pode se relacionar com no mínimo 1 e, no máximo, 7 instancias da classe Professor e que cada instância da classe Professor se relaciona com no mínimo zero e, no máximo, muitas instancias da classe Aluno.

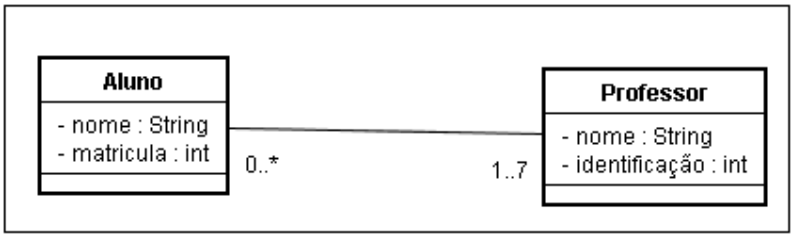

Figura 2.8. Associação entre duas classes

A Composição é um tipo de relacionamento que indica que os objetos de uma classe, ou objetos-parte, devem pertencer única e exclusivamente a um único objeto de outra classe, ou objeto-todo. Nesse tipo de relacionamento, quando um objeto-todo deixa de existir, seus objetos-parte também deixam, porém, o contrário não é verdade. A figura 2.9 ilustra um exemplo de Composição.

Como pode-se observar na figura 2.9, cada objeto da classe Carro contém no mínimo 1 e, no máximo, 4 objetos da classe Pneu e cada objeto deste, se relaciona a apenas uma instância da classe Carro. Sendo o relacionamento do tipo composição, ao se eliminar algum objeto da classe Carro, os objetos da classe Pneu que se relacionam a ele também são eliminados.

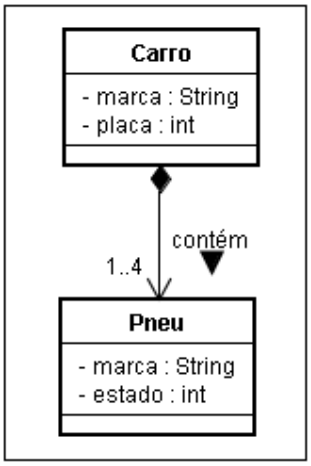

Figura 2.9. Composição

No relacionamento do tipo Especialização/Generalização ocorre a identificação das classes-mães, chamadas gerais, e das classes-filhas, chamadas de especializadas. Ela ocorre quando existe duas ou mais classes com características semelhantes. Dessa forma, é criada uma classe mais geral que possui os atributos e métodos em comum e são declaradas classes especializadas ligadas a classe geral. Por meio desse mecanismo é possível o reaproveitamento de código, pois as classes filhas irão herdar os métodos e atributos da classe mãe, além de possuírem os seus próprios métodos e atributos. Vale ressaltar que os métodos da classe-mãe podem ser redeclarados nas classes filhas para que realizem funções diferentes daquelas descritas anteriormente. Portanto não é preciso modificar os métodos mais gerais caso eles sejam indesejáveis. Basta modificá-los nas próprias classes especializadas.

A figura 2.10 mostra um exemplo de relacionamento do tipo Especialização/Generalização. Pela figura, observa-se a existência de uma classe geral chamada Carro e de duas classes especializadas chamadas Carro de Passeio e Carro de Estrada. Estas possuem seus próprios atributos e métodos, porém, herdam os atributos e métodos da classe-mãe, Carro.

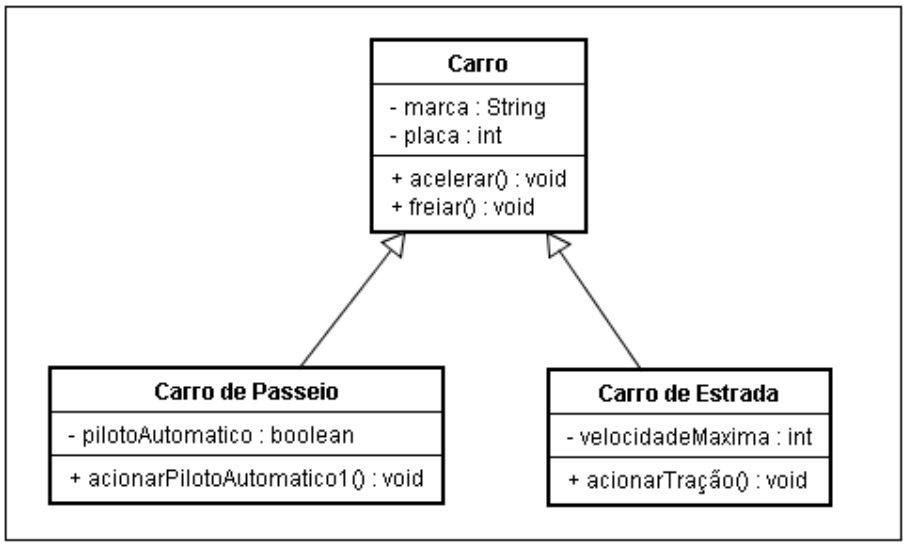

Figura 2.10. Relacionamento do tipo Especialização/Generalização

2.1.3. Diagrama de Sequência

O Diagrama de Sequência enfoca a ordem temporal em que as mensagens são trocadas entre os objetos durante determinado processo. Cada diagrama de sequência se relaciona, em geral, a um diagrama de casos de uso específico, compondo, portanto, parte da documentação desse caso de uso. O seu objetivo principal é mostrar como ocorre a interação entre os objetos a partir de uma sequência ordenada de eventos. É mostrado as mensagens que são trocadas e os métodos disparados para que determinado processo seja concluído. Ele se relaciona com o diagrama de classes, pois utiliza tanto os métodos quanto os objetos referentes às classes instanciadas de tal diagrama.

Os diagramas de sequência se iniciam, em geral, com alguma mensagem ou estímulo vindo de um Ator para algum objeto do sistema. Neste momento, passa a ocorrer uma série de chamadas de métodos de forma sequencial entre objetos específicos do software até que o processo seja plenamente executado.

Um componente importante presente no diagrama de sequência é a Linha de Vida. Ela representa o tempo em que um objeto existiu durante um processo. É ilustrada por linhas finas verticais tracejadas partindo do retângulo que representa o objeto. Outro conceito essencial é o Foco de Controle que indica os períodos em que determinado objeto está participando ativamente do processo. Eles são representados dentro da Linha de Vida por uma linha mais grossa. A figura 2.11 mostra um exemplo de Diagrama de Sequência, na qual observa-se as indicações da Linha de Vida e do Foco de Controle, como descrito anteriormente. O componente utilizado para realizar essa indicação é próprio da linguagem UML e chama-se Nota. Na figura, o ator Aluno forçou o disparo do método VerificarDados() do objeto matemática referente à classe Matéria. A seta tracejada com o nome Dados do Aluno representa uma mensagem de retorno e, neste caso, representa informações especificas do método chamado. As mensagens de retorno muitas vezes retornam apenas um valor indicando se o método foi realizado com sucesso ou não. Observa-se também que o objeto matemática realiza o disparo do método CalculaNota() em si mesmo. Esse tipo de mensagem é denominada auto-chamadas. Apesar de não estar mostrado no exemplo, normalmente as mensagens ocorrem entre um objeto e outro objeto. Neste caso, um objeto transmite uma mensagem para o outro solicitando a execução de um método.

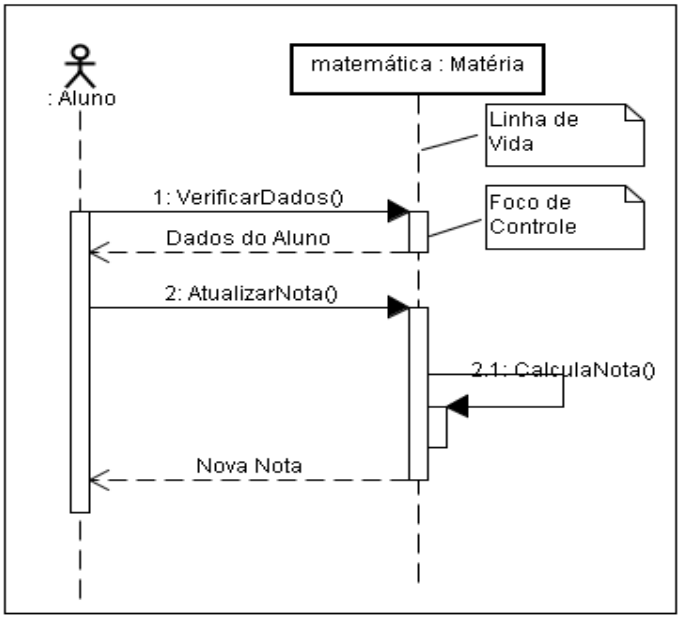

Figura 2.11. Diagrama de Sequência

2.2. Desenvolvimento Iterativo e Incremental

O desenvolvimento iterativo e incremental é um processo de desenvolvimento de software. Ele surgiu pela combinação das estratégias do desenvolvimento incremental e do desenvolvimento iterativo. O desenvolvimento incremental estabelece que várias partes do sistema devem ser desenvolvidas em paralelo e, ao final, integradas, enquanto que o desenvolvimento iterativo planeja o retrabalho sobre o processo de produção e define o tempo de revisão e de melhorias de parte do sistema [3].

O processo se inicia com o estabelecimento de requisitos básicos do software. A partir do ponto em que o projeto se desenvolve, os requisitos são aumentados gradativamente, de forma iterativa até que o software se desenvolva plenamente. A cada iteração o projeto é modificado conforme necessário e são adicionadas novas funcionalidades ao programa.

Todo o desenvolvimento pode ser resumido na etapa de inicialização, de iteração e na lista de controle do projeto. Na primeira etapa, é criada a versão fundamental do software de forma a criar um produto que possa ser avaliado e testado tanto pelo desenvolvedor quanto pelo futuro usuário do sistema. Este produto base deve exemplificar aspectos fundamentais do programa e deve ser simples o bastante para que possa ser compreendido sem muitas dificuldades. A lista de controle de projeto é um registro de todas as tarefas que devem ser feitas, incluindo as novas funcionalidades a serem incluídas. Ela serve como um guia para o desenvolvimento e deve ser revisada sempre. Na etapa de iteração deve-se analisar a versão corrente do software e codificar as tarefas da lista de controle de projeto. Ao final de cada iteração deve ser feita uma análise dos aspectos adicionados em relação a modularidade, usabilidade, reusabilidade, eficiência e alcance dos objetivos. Deve-se então revisar e atualizar a lista de controle. Um dos principais padrões de sistemas de desenvolvimento iterativos é o Processo Unificado da Rational.

2.2.1. Processo Unificado da Rational

O Processo Unificado da Rational, denominado RUP, do inglês, *Rational Unified Process*, foi criado pela *Rational Software Corporation*, posteriormente adquirida pela IBM. Sua meta é garantir a produção de software de alta qualidade que atenda às necessidades dos usuários dentro de um cronograma e de um orçamento previsíveis. Muitas de suas características foram formadas a partir de práticas adotadas comercialmente

e cujas eficiências foram comprovadas na prática. É um processo melhor aplicado a grandes equipes de desenvolvimento e a projetos complexos. Porém, muitas de suas características podem ser adaptadas e adequadas para o desenvolvimento de sistemas de pequeno e médio porte.

A arquitetura geral do RUP está mostrada na figura 2.12 [4]. O eixo horizontal representa o tempo e mostra os aspectos do ciclo de vida do software na medida em que se desenvolve. Como pode ser visto, o tempo é medido em fases e iterações. O eixo vertical representa as disciplinas, que agrupam as atividades de maneira lógica. Os gráficos internos mostram como é a intensidade das atividades durante o desenvolvimento do software. Por exemplo, na fase de inicialização existe uma grande atividade relacionada com a modelagem de negócios, enquanto que, na fase de construção, essa atividade é bem reduzida.

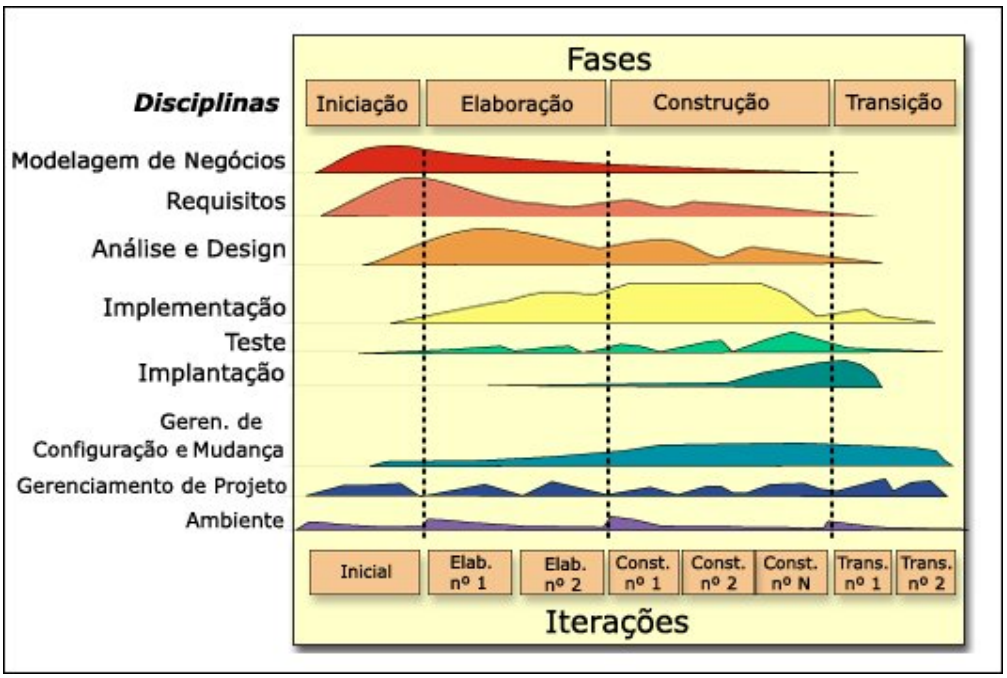

Figura 2.12. Arquitetura geral do RUP

A meta dominante da fase de iniciação é atingir o consenso entre todos os envolvidos sobre os objetivos do ciclo de vida do projeto [4]. É uma fase bastante importante, principalmente para os novos desenvolvimentos, em que os requisitos precisam ser tratados adequadamente para evitar complicações futuras. Para os projetos que buscam atualizar e melhorar sistemas já existentes, a fase de inicialização é mais rápida, mas, nem por isso,

menos importante. É nessa fase que se devem elaborar os principais casos de uso do sistema, estimar os custos gerais, estimar os riscos potenciais e preparar o ambiente de suporte para o projeto.

A fase de elaboração tem como principal objetivo fornecer uma base estável para arquitetura do sistema. Esta arquitetura deve se desenvolver a partir dos requisitos mais significativos e de uma avaliação de risco. Deve-se, nessa fase, assegurar que a arquitetura e os requisitos sejam estáveis o suficiente para que os riscos sejam diminuídos a fim de se estimar com relativa precisão os custos do projeto e a quantidade de programação necessária para a conclusão do mesmo. Deve-se também demonstrar que a arquitetura base suportará os requisitos a um custo e tempo justos.

O objetivo da fase de construção é esclarecer os requisitos restantes e determinar, à partir da arquitetura base, o desenvolvimento do software. A ênfase, nessa fase, está no controle e gerenciamento das operações de forma a otimizar custos, qualidade e quantidade de programação. Devem-se minimizar os custos evitando retrabalho e otimizando recursos, atingir a qualidade adequada com rapidez e eficiência, concluir os testes de todas as funcionalidades necessárias e, finalmente, desenvolver de maneira iterativa um produto completo, que esteja pronto para o usuário. É fundamental a existência de paralelismo entre o trabalho das equipes de desenvolvimento de forma a acelerar as atividades.

A fase de transição assegura que o software esteja disponível para os usuários finais. Esta fase pode passar por inúmeras iterações que incluem ajustes finos no produto com base na resposta do usuário. Ela pode ser simples, mas também bastante complexa, a depender das mudanças e correções que deverão ser feitas. Inclui, nessa fase, o treinamento de usuários e equipes de manutenção, introdução ao marketing das vendas, conversão de banco de dados operacionais etc.

Toda descrição do RUP é mostrada de maneira completa e detalhada na referência [4], na qual são fornecidos diversos diagramas e diretrizes que devem ser obedecidas no desenvolvimento do programa.

2.3. Padrões de Projeto

Padrões de Projeto, denominados *Design Patterns*, em inglês, descrevem soluções para problemas recorrentes relacionados ao desenvolvimento de software que utilizam o paradigma da orientação a objetos. Esses padrões constituem um repertório de princípios

gerais de soluções idiomáticas que guiam os projetistas na criação de software. Eles são codificados de forma estruturada, mostram um problema aliado a uma solução e são identificados por um nome. Buscam simplificar a reutilização de soluções durante o projeto do software.

Os padrões não pretendem expressar novos princípios de projeto. Na verdade, trata-se exatamente do oposto, eles tentam codificar conhecimento, idiomas e princípios existentes, testados e consagrados; quanto mais apurados e amplamente usados, melhor [5]. Os dois conjunto de padrões mais amplamente utilizados no projeto de softwares são GoF (*Gang of Four*) e GRASP (*General Responsability Assignment Software Patters*).

Os GoF são divididos em padrões de criação, estruturais e comportamentais. Os padrões de criação referem-se ao estabelecimento das classes e objetos do software. Os padrões estruturais se relacionam com a associação existente entre as classes e objetos; enquanto que os comportamentais se relacionam com a divisão de responsabilidades entre esses dois elementos e a organização deles dentro do sistema. Na Tabela 2.1 está mostrado os padrões GoF em seus grupos específicos.

Tabela 2.2. Padrões GoF

Padrões de Criação	**Padrões Estruturais**	**Padrões Comportamentais**
Abstract Factory	*Adapter*	*Chain of Responsability*
Builder	*Bridge*	*Command*
Factory Method	*Composite*	*Interpreter*
Prototype	*Decorator*	*Iterator*
Singleton	*Facade*	*Mediator*
	Flyweight	*Memento*
	Proxy	*Observer*
		State
		Strategy
		Template Method
		Visitor

Os padrões GRASP descrevem princípios fundamentais de projeto baseado em objetos e atribuição de responsabilidades aos mesmos [5]. Tais princípios são a base de um

projeto de sistema e devem ser bem dominados por qualquer desenvolvedor. Os padrões GRASP estão mostrados na Tabela 2.2.

Tabela 2.3. Padrões GRASP

Padrões GRASP
Controller
Creator
Expert
Law of Demeter
Low Coupling/High Cohesion
Polymorphism
Pure Fabrication

Dois importantes padrões, que serão utilizados na elaboração do programa, são *Singleton* e *Mediator*. O padrão *Singleton* assegura a existência de apenas uma instância de determinada classe durante a execução do programa. O padrão *Mediator*, por sua vez, auxilia a comunicação entre os objetos, centralizando-a em um único componente.

2.4. Linguagem de programação utilizada

A linguagem de programação utilizada para confeccionar o software FDTD Studio é a C++. Esta é uma linguagem desenvolvida a partir da linguagem de programação C, cujo objetivo principlal era subtituir a programação em *assembler* nas tarefas de programação de sistemas mais exigentes. O C++ foi projetado de forma a não se compremeter os ganhos obtido na linguagem C, além de se tornar mais segura, expressiva e de diminuir a necessidade de se utilizar técnicas de baixo nível. Com o C++, é possível que grandes programas sejam estruturados de forma racional, tornando possível para um único indivíduo a trabalhar com uma grande quantidade de código. Os mecanismos disponíveis no C++ fornecem segurança e facilidade de compreensão do código pelos desenvolvedores. Além disso, eles foram destinados a lidar com recursos de hardware de modo eficiente e direto, possuindo ainda recursos para ocultar toda a base de código sob interfaces seguras e bem estruturadas. O C++ possui qualidades como modularidade,

interfaces fortemente tipadas e flexibilidade, o que o torna adequado para a escrita de grandes programas, onde se espara a cooperação conjuta de vários grupos de programadores.

Uma linguagem de programação serve para dois objetivos correlatos: fornece um vínculo para o programador especificar ações a serem executadas e fornece um conjunto de conceitos para o programador usar quando pensa sobre o que deve ser feito [6]. O primeiro objetivo pressupõe a existência de mecanismos de programação próximos da máquina e que possam ser utilizados de maneira rápida e prática. A linguagem de programação C foi designada especialmente para isso. O segundo objetivo, por sua vez, requer recursos que sejam próximos do problema a ser resolvido. A linguagem C++ veio para suprir essa necessidade.

O C++ possui irrestrita eficiência em baixo nível o que o torna adequado para escrita de controladores de dispositivos e softwares que dependam da interação direta com o hardware e sujeitos a restrições de tempo real. Ela foi projetada de forma que cada um de seus recursos possa ser utilizado em códigos sujeitos a severas restrições de tempo.

Grande parte dos softwares possui estruturas que são essenciais para um desempenho aceitável. Entretanto, a maioria do código dos programas não faz parte de tais estruturas e é fundamental que possuam qualidades como manutenibilidade, facilidade de testes e facilidade de extensão. O C++ visa cobrir essas necessidades e isso fez com que seu uso fosse generalizado ao longo do tempo, sobretudo em aplicações que exijam confiabilidade e cujos requisitos mudam frequentemente tais como sistemas bancários, comerciais, militares e de telecomunicações.

Outras áreas em que é forte a utilização do C++ são as que necessitam de interfaces robustas com usuários e que se utilizam extensivamente de gráficos que auxiliam na análise de computações numéricas, científicas e de engenharia. Assim, a principal capacidade do C++ é ser eficiente e adequado nas mais diversas áreas de aplicação e que exijam um número grande de estruturas inter-relacionadas como computações numéricas, gráficos, interação com o usuário, acesso a base de dados, sistemas de redes locais e de longa distância etc. O C++, sobretudo, está apto a coexistir com trechos de códigos escritos em outra linguagem, o que aumenta ainda mais o leque de aplicações.

2.5. Ambiente de desenvolvimento integrado utilizado

Ambiente de desenvolvimento integrado, IDE, do inglês, *Integrated Development Environment* é um programa que possui ferramentas e dispositivos que visem agilizar a produção de softwares.

A IDE escolhida para o desenvolvimento da nova versão do FDTD Studio foi o Code Blocks, um ambiente aberto e multiplataforma, que ainda está em desenvolvimento. Esta IDE, feita em C++, usa wxWidgets para a contrução de sua própria interface gráfica. Ela é voltada, exclusivamente, para o desenvolvimento em C/C++ e fornece suporte para diversos compiladores. Também suporta wxSmith, uma ferramenta RAD (*Rapid Application Development*) para wxWidgets.

3. ANÁLISE DO SOFTWARE FDTD STUDIO

Como dito anteriormente, o FDTD Studio é um software para análise e projeto de antenas. A sua principal característica é a utilização do Método das Diferenças Finitas no Domínio do Tempo (*Finite Difference Time Domain* – FDTD), um método numérico usado para solucionar as equações diferenciais parciais de Maxwell. Este método, como o próprio nome indica, baseia-se na solução das equações diretamente no domínio do tempo, o que proporciona características vantajosas tais como: número de incógnitas ilimitados, dependentes apenas, da capacidade de memória da máquina; fontes de erro conhecidas e que podem ser adequadamente manipuladas; cálculos diretos da resposta não–linear de um sistema eletromagnético; visualização dinâmica dos campos eletromagnéticos ao longo do tempo etc [1]. É um software bastante completo que permite de maneira simples e rápida a modelagem de diversas estruturas diferentes.

A figura 3.1 ilustra a tela principal do programa. Como pode ser observada, a interface é composta, basicamente, por quatro áreas separadas. Na parte superior, existe a barra de tarefas com atalhos para as funcionalidades abrir, salvar e novo projeto, além de outras ferramentas utilizadas para manipular a estrutura em análise, simular o projeto e para exibir os resultados. Na parte central é exibido o domínio computacional, onde é visualizado o sistema em análise. A parte da esquerda é subdividida em duas janelas distintas, a superior, denominada *Project*, onde se encontram os objetos nativos do domínio computacional e aqueles que vierem a ser inseridos no domínio, e a inferior, denominada *Objects*, onde se encontram os objetos que podem ser inseridos no domínio computacional. Estes estão divididos em quatro conjuntos específicos denominados *Objects*, *Sources*, *Boundaries* e *Probes*. Na parte inferior da tela principal existe uma caixa de texto onde são exibidas informações úteis como erros, alertas, estados etc.

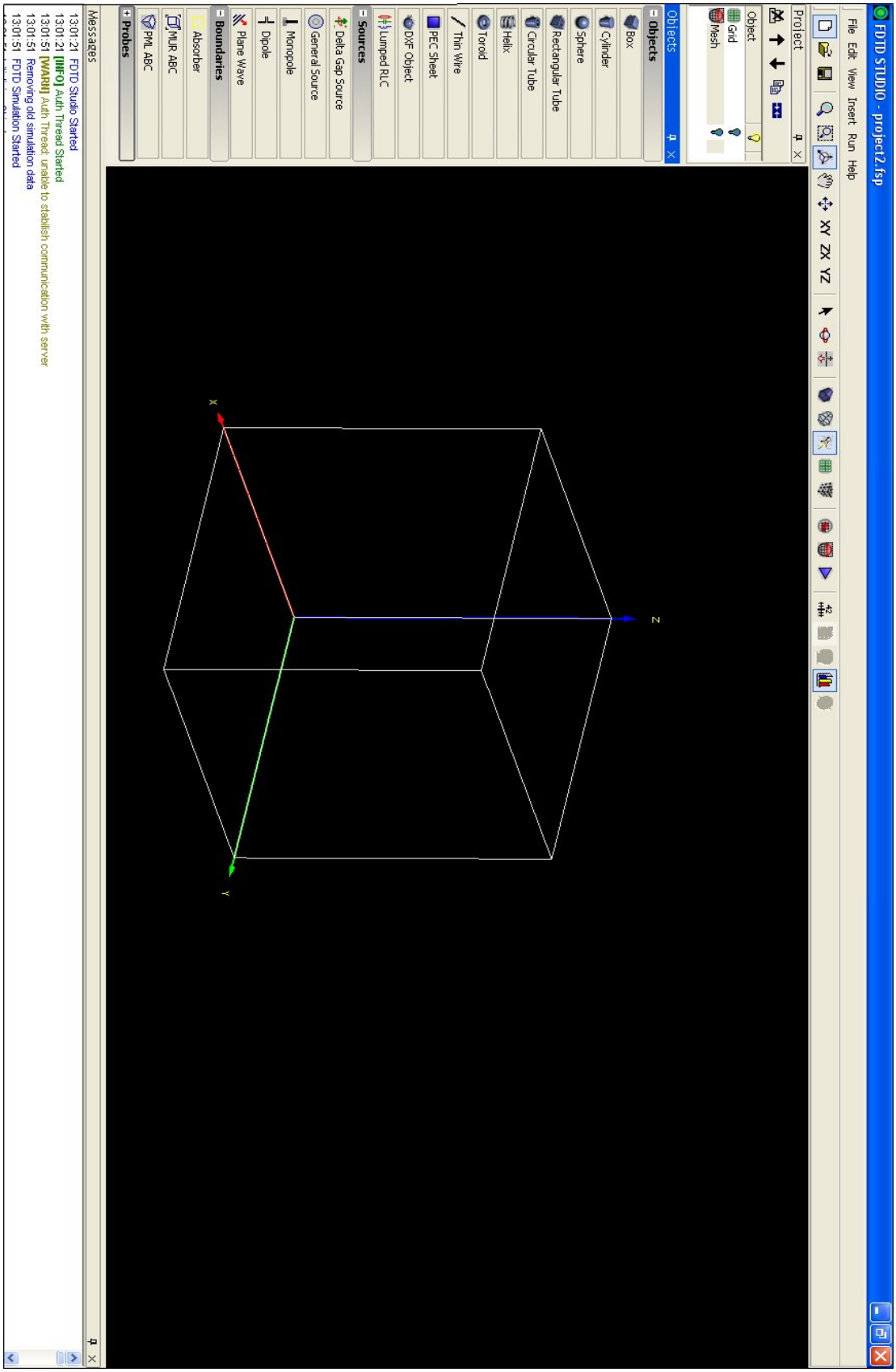

Figura 3.1. Tela principal do programa

O objeto Grid, localizado na janela *Project*, é o principal elemento do domínio computacional. Ele é responsável pela determinação das características da malha computacional, do intervalo de tempo e do número de intervalos que devem ser utilizados na simulação. Ao clicar em seu ícone, é aberta uma nova janela, mostrada na figura 3.2, na qual é possível configurar as posições mínimas e máximas das células de cada eixo e os respectivos números de células. Pode-se configurar, também, o espaçamento mínimo entre as células, o número de intervalos de tempo e o valor de cada intervalo.

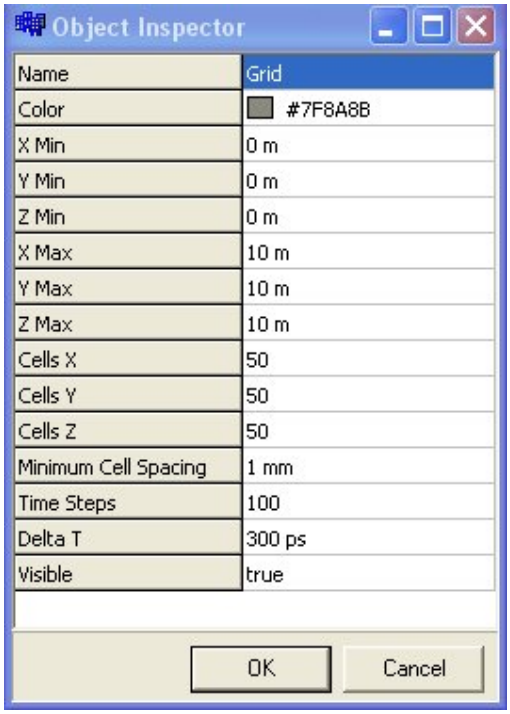

Figura 3.2. Janela de configuração do grid

Para ilustrar a simulação de um novo projeto, será inserido no domínio computacional, um objeto fonte do tipo dipolo. Clicando no ícone *Dipole* na janela *Objects*, é aberta a janela mostrada na figura 3.3. Os parâmetros dessa janela são utilizados para configurar a antena dipolo a ser inserida no domínio. Clicando no botão *OK*, é inserida na malha computacional a antena dipolo requerida. A figura 3.4 ilustra o novo domínio computacional, já com a inserção da antena dipolo. Pode-se observar que a antena dipolo foi adicionada no domínio com as configurações *defaut* de sua janela.

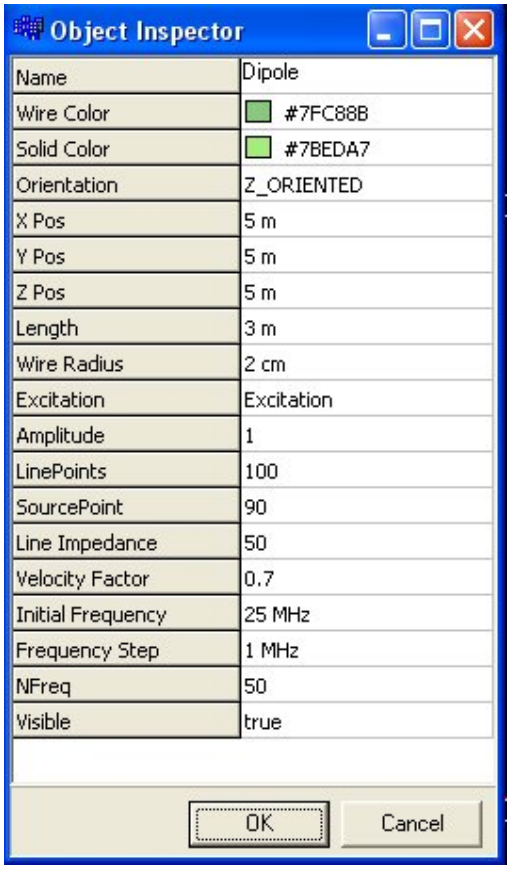

Figura 3.3. Janela de configuração do objeto *Dipole*

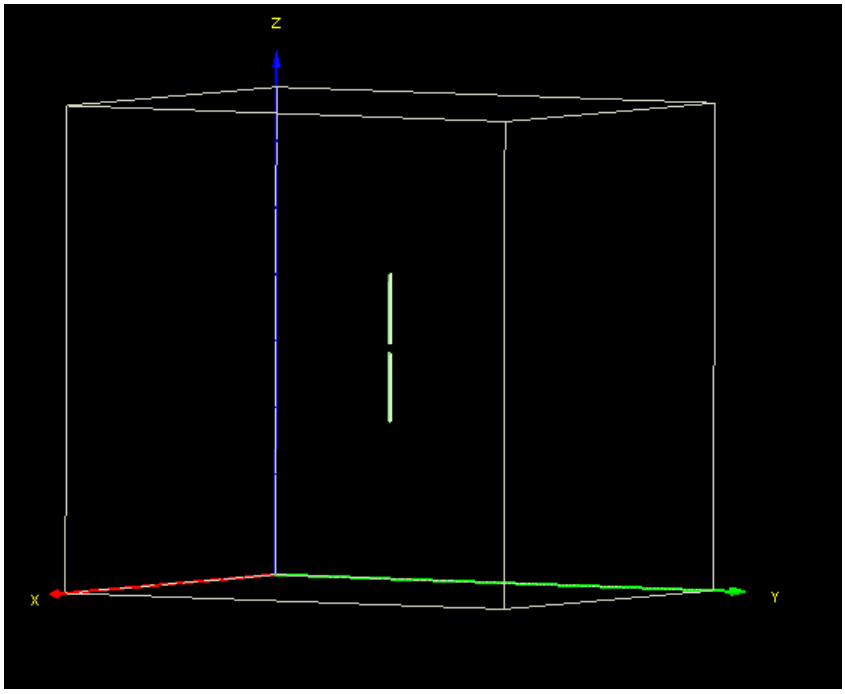

Figura 3.4. Domínio computacional com antena dipolo

Para simular este projeto, bata clicar no ícone na barra de ferramentas. É aberta, então, uma janela de confirmação da simulação mostrada na figura 3.5. Após clicar em *Yes*, é preciso salvar o projeto com um nome desejado. A simulação então se inicia automaticamente. Durante a simulação, é mostrada uma barra indicando o seu progresso. A figura 3.6 ilustra essa barra no instante em que 26% da simulação já foi concluída.

Figura 3.5. Janela de confirmação da simulação

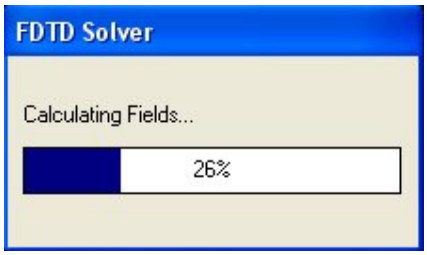

Figura 3.6. Barra indicando a conclusão de 26% da simulação

Ao término da simulação, é preciso clicar no ícone da barra de tarefas para que os resultados possam ser exibidos. É aberta então uma janela denominada *Results*, onde se podem selecionar os resultados desejados. A figura 3.7 ilustra essa janela com a opção *Reflected Voltage* selecionada. Como pode ser visto, é exibido um gráfico de mesmo nome com os valores da tensão refletida da fonte do tipo dipolo. Se existisse, por exemplo, mais de uma fonte na malha computacional, na janela *Results*, iria existir, além do guia *Dipole*, o guia referente à outra fonte do domínio, com os dados respectivos.

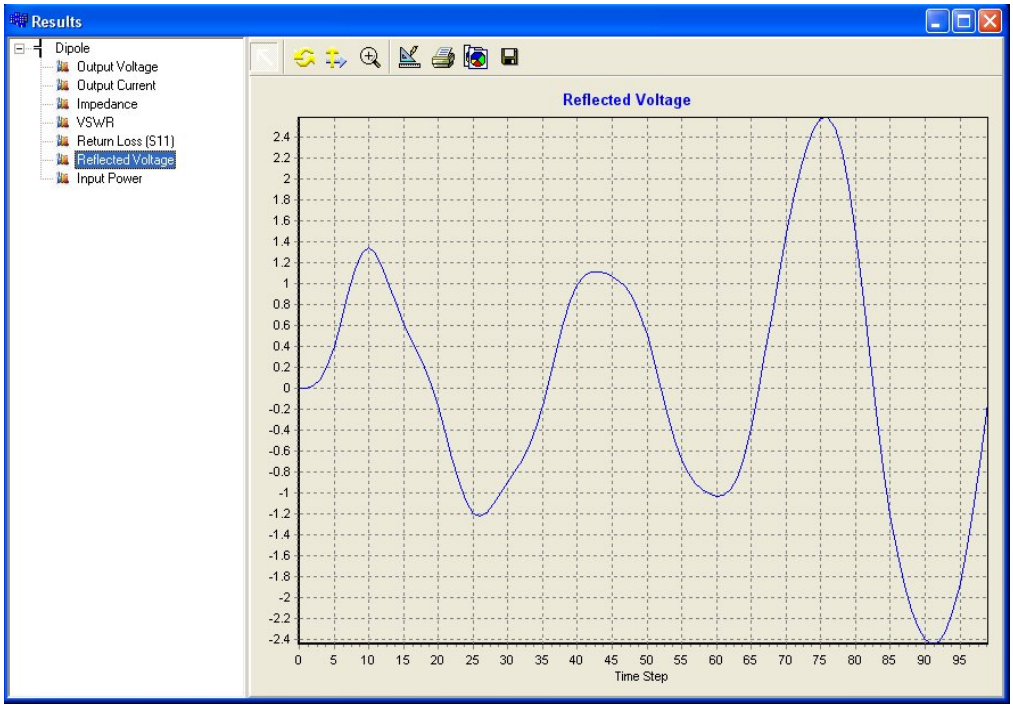

Figura 3.7. Janela *Results* com os dados da tensão refletida da antena dipolo

3.1. Estrutura básica do programa

A estrutura básica do software FDTD Studio é encabeçada por uma classe mãe denominada TFDTDObject. Todos os outros objetos referentes ao processamento básico do programa são objetos filhos dessa classe. Tal classe possui inúmeros métodos e atributos que são herdados pelas classes filhas. Porém, nem todos os métodos e atributos são utilizados por todas as classes herdadas já que existem classes filhas que não necessitam dos métodos ou atributos descritos.

A utilização da classe básica TFDTDObject é essencial para que seja utilizado o processamento polimórfico nas classes do programa. Isso ocorre já que todos os outros objetos podem ser tratados como TFDTDObject e, assim, o processamento pode ocorrer em todos os objetos desejados a partir de um loop comum de ponteiros à classe TFDTDObject. Essa característica é utilizada em grande parte do processamento do software, seja no cálculo dos campos ou na exibição dos gráficos resultantes. A figura 3.8 ilustra essa classe com alguns de seus métodos e atributos.

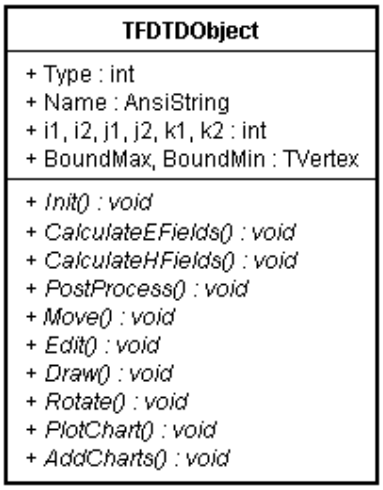

Figura 3.8. Classe TFDTDObject

Como pode-se observar, todos os métodos mostrados na figura 3.8 são abstratos (*itálico*). Em C++ os métodos abstratos são também denominados virtuais. Tais métodos são responsáveis pelo comportamento polimórfico dos objetos, pois, com eles, o tipo de objeto apontado pelo ponteiro determina qual versão da função virtual chamar [7].

Os atributos Type e Name representam o tipo e o nome do objeto, respectivamente. Os atributos BoundMax e BoundMin representam a maior e a menor dimensão, enquanto que os atributos i1, i2, j1, j2 k1 e k2 indicam os pontos extremos do grid ocupados pelo objeto.

O método *Init()* é utilizado para inicializar o objeto antes do cálculo dos campos. Os métodos *CalculateEFields()* e *CalculateHFields()* são utilizados, respectivamente, para o cálculos dos campos elétrico e magnético do objeto. O método *PostProcess()* é utilizado para realizar um pós-processamento após o cálculo dos campos. Os métodos *Move()* e *Rotate()* são utilizados, respectivamente, para mover e rotacionar o objeto. O método *Edit()* é utilizado para abrir a tela de edição dos parâmetros do objeto. O método *Draw()* é chamado para desenhar o objeto na tela de interação com o usuário. Por fim, os métodos *AddCharts()* e *PlotCharts()* são chamados, respectivamente, para adicionar os gráficos e plotar os gráficos resultantes do cálculo dos campos.

Como é possível observar no código fonte do programa, esses métodos são apenas protótipos de funções sem definição alguma. Isso é feito, pois não faz sentido a chamada de algum desses métodos sem se ter a instanciação de um tipo de objeto específico. Por

exemplo, não é possível a chamada dos métodos *Draw()* ou *Init()* de um TFDTDObject genérico. É preciso saber quais objetos deseja-se desenhar ou inicializar.

A partir de TFDTDObject, são gerados, por herança, todos os principais tipos de classes do programa, sobretudo, TSource, TObject3D, TMesh, TGrid, TProject e classes de fronteira como TPmlABC.

3.2. A classe TSouce

A classe TSource é uma classe mãe de todas as fontes do programa. Nela, são redeclarados e definidos alguns dos métodos presentes em TFDTDObject tais como *Move()* e *Init()*. É preciso lembrar que, apesar desses métodos serem definidos em TSource, eles podem ser redefinidos nas classes filhas, caso seja conveniente como é observado no código fonte do programa. Outros métodos como *CalculateEFields()*, *CalculateHFields()* e *PostProcess()* não são definidos em TSource, já que é preciso saber qual tipo de fonte é utilizada e quais seus parâmetros para que possam ser efetivamente descritos. Na figura 3.9 está ilustrado a classe TSource com alguns de seus métodos e atributos.

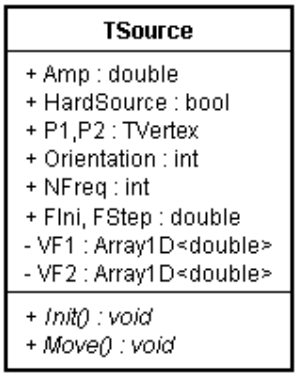

Figura 3.9. Classe TSource

Uma classe que herda diretamente de TSource é TDipole, que representa a fonte do tipo dipolo. A figura 3.10 ilustra essa classe. Como pode ser observado, essa classe redeclara alguns dos métodos mostrados em TFDTDObject. É nessa classe que são definidos também os métodos *CalculateEFields(), CalculateHFields(), AddCharts(), PlotChart(), Edit(), Draw()* e *Rotate()*. Faz sentido definir tais métodos nessa classe, pois

já haverá uma fonte específica em que os parâmetros são definidos para o cálculo dos campos e posterior plotagem dos resultados. Pode-se observar, também, que, apesar dos métodos *Move()* e *Init()* já terem sidos definidos em TSource, eles foram redefinidos em TDipole. Essa redefinição é interessante do ponto de vista polimórfico, já que, em uma chamada desse tipo, será dada prioridade ao método redefinido na fonte, porém, caso esse método não exista, haverá um método *defaut* a ser chamado em TSource.

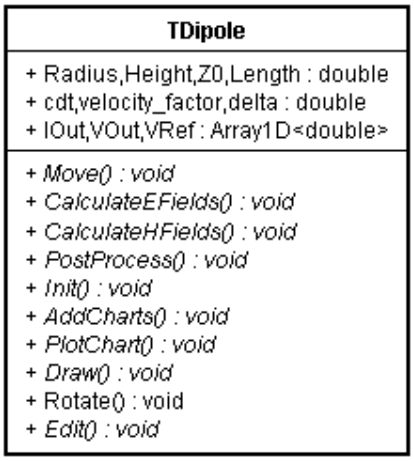

Figura 3.10. Classe TDipole

3.3. A classe TPmlABC

A classe TPmlABC representa a fronteira de absorção do tipo PML (*Perfect Matched Layer)*. A figura 3.11 ilustra tal classe com seus métodos e atributos principais. Como pode ser observado no código fonte, essa classe redeclara e define os métodos *Edit(), Init(), CalculateEFields()* e *CalculateHFields()* de sua classe mãe, TFDTDObject. Pode-se notar, também, que não são definidos os métodos referentes aos gráficos nessa classe já que interessa apenas os dados relativos às antenas propriamente ditas.

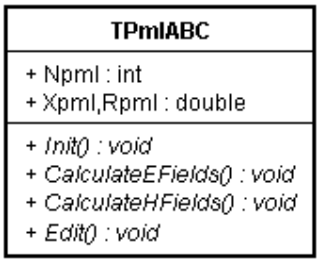

Figura 3.11. Classe TPmlABC

3.4. A classe TObject3D

A classe TObject3D é a classe mãe de todos os objetos tridimensionais que podem ser colocados dentro da malha computacional, com exceção das fontes que são definidas em Tsource. A figura 3.12 ilustra essa classe e alguns de seus métodos e atributos. Essa classe redeclara e define os métodos *Move(), Rotate(), Init(), Draw()* e *Edit()* de sua classe mãe, TFDTDObject. Ela também possui como atributos um vetor de ponteiros a elementos TFacet, que armazena as faces que compõe o objeto, e um elemento do tipo TMaterial, que armazena características tais como permissividade, permeabilidade, condutividade, perda magnética e densidade específica do material.

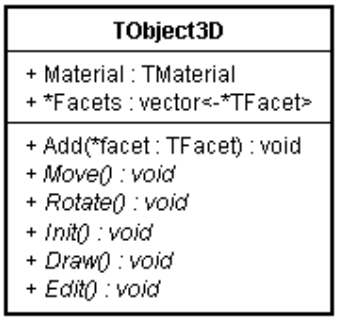

Figura 3.12. Classe TObject3D

Uma classe que herda diretamente de TObject3D é TBox, ilustrada na figura 3.13. Da mesma forma em que TDipole, essa classe define novamente os métodos *Move(), Rotate()* e *Edit()* definidos em sua classe mãe. Como em TPmlABC, não ocorre a definição dos métodos de exibição dos dados, já que interessa apenas aqueles relativos às antenas.

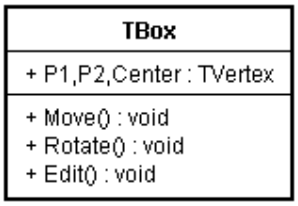

Figura 3.13. Classe TBox

3.5. A classe TGrid

A classe TGrid, ilustrada na figura 3.14, armazena os parâmetros característicos da malha tridimensional, tais como número de células em cada dimensão, dimensões das células e outros como variação do tempo, número de intervalos de tempo, maior freqüência, etc. Nela, também, é definido os métodos *Init()*, *Edit()* e *Draw()* de sua classe mãe.

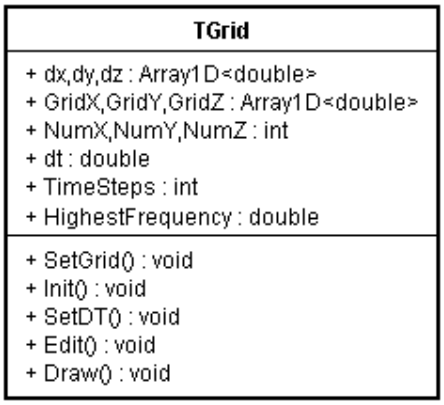

Figura 3.14. Classe TGrid

3.6. A classe TMesh

A classe TMesh, figura 3.15, determina parâmetros fundamentais da malha tais como as características elétricas e magnéticas de cada célula, representada pelo vetor tridimensional Mat, e os coeficientes *Ca*, *Cb*, *Da* e *Db*, relacionados ao material inserido na malha. Estes coeficientes são parâmetros de uma classe específica denominada TMaterialCoefficient. Em TMesh, o vetor LookUpTable armazena referências a elementos dessa classe. O vetor tridimensional Coefficients armazena inteiros que se relacionam com cada posição do vetor LookUpTable. Tal vetor tridimensional representa a malha em si, assim, cada célula da malha será representada por um inteiro que se relaciona a um TMaterialCoefficient. A determinação dos elementos do vetor Mat é feita pelo método do cruzamento dos raios [1]. Este método é implementado na função-membro Generate() (*ver código fonte*). A partir do vetor Mat, é determinado os elementos do vetor LookUpTable e do vetor tridimensional Coefficient pela função-membro GetCoefficients() (*ver código fonte*). Em TMesh também ocorre a definição dos métodos *Draw()* e *Edit()*.

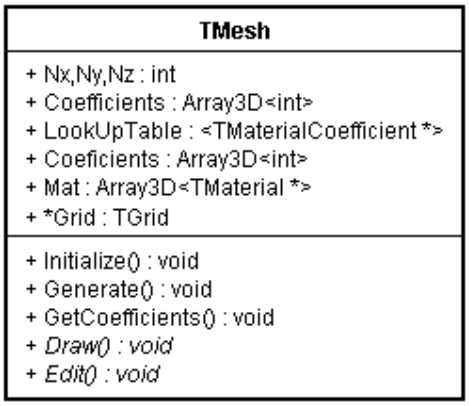

Figura 3.15. Classe TMesh

O procedimento utilizado em TMesh para armazenar os parâmetros *Ca*, *Cb*, *Da* e *Db* foi útil para que se pudesse economizar memória física durante a simulação. Suponha uma célula bidimensional, no plano xy, com os respectivos vetores de campos elétrico e magnético, representada na figura 3.16. Suponha, agora, uma malha bidimensional, composta de várias dessas células, figura 3.17. Cada cor representa um tipo de material diferente. Pode-se observar, no codigo fonte de TMesh, que a dimensão de cada componente do vetor tridimensional Mat é igual ao número de células nas dimensões respectivas. Porém, o vetor Coefficients possui uma dimensão a mais em cada uma de suas componentes. Isso é feito pois é necessário componentes de campos elétrico e magnético a mais para fechar o domínio computacional [1]. Assim, a malha da figura 3.17 pode ser representada como na figura 3.18 que ilustra essa dimensão a mais nas componentes do vetor Coefficients. Portanto, o vetor Coefficients pode ser visto como uma representação da malha tridimensional, porém, com uma camada de células a mais nos limites superiores dos planos x, y e z.

Pode-se dividir a malha bidimensional da figura 3.18 em agrupamentos de células que terão o conjunto de coeficientes *Ca*, *Cb*, *Da* e *Db* semelhantes. Essa divisão é feita na figura 3.19, na qual observa-se 27 agrupamentos de células, cada um com um mesmo conjunto de coeficientes compondo suas células. Portanto, será acrescido 27 elementos ao vetor LookUpTable, cada um representando coeficientes *Ca*, *Cb*, *Da* e *Db* distintos. O vetor tridimensional de inteiros Coefficients, por sua vez, terá uma variedade de 27 inteiros diferentes em sua composição, cada um se relacionando com uma posição do vetor LookpTable distinta. Dessa forma, economiza-se memória armazenando apenas 27 conjuntos de coeficientes no lugar de um conjunto para cada célula da malha. A situação é

bem mais complexa em uma malha tridimensional com um número grande de materiais diferentes, entretanto o benefício obtido por meio desse procedimento se repete.

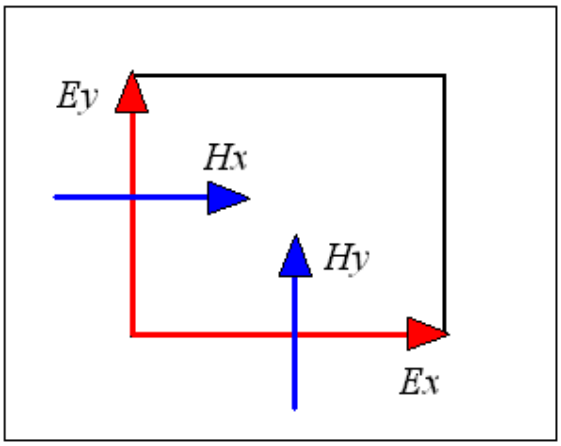

Figura 3.16. Célula bidimensional

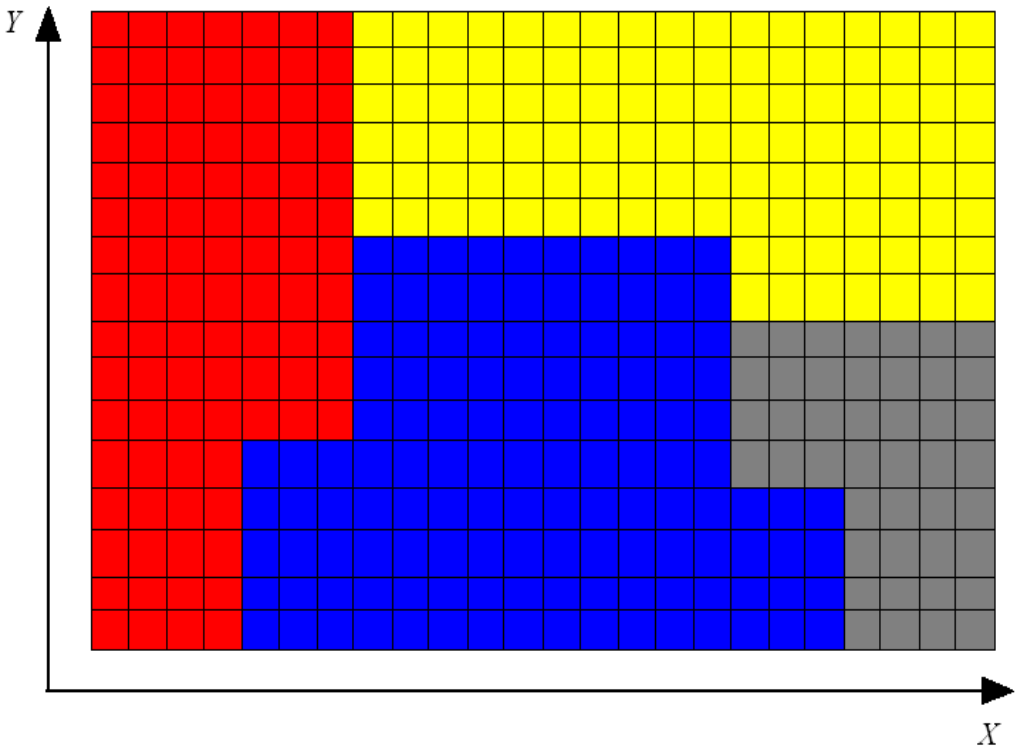

Figura 3.17. Malha bidimensional

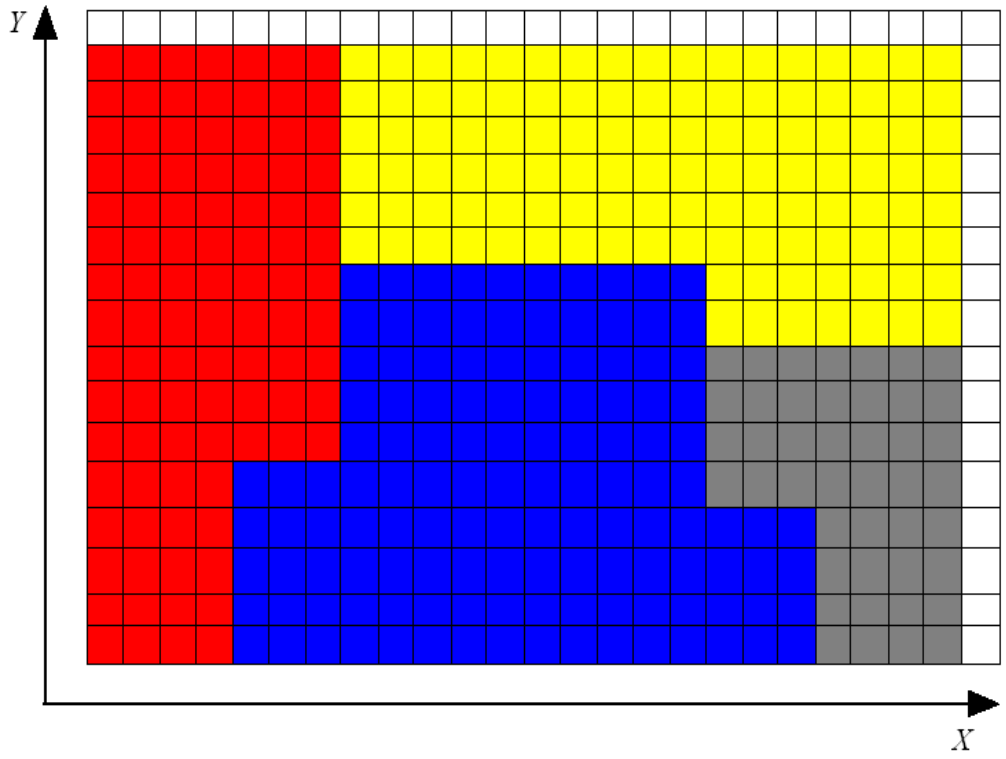

Figura 3.18. Malha bidimensional com camadas de células a mais

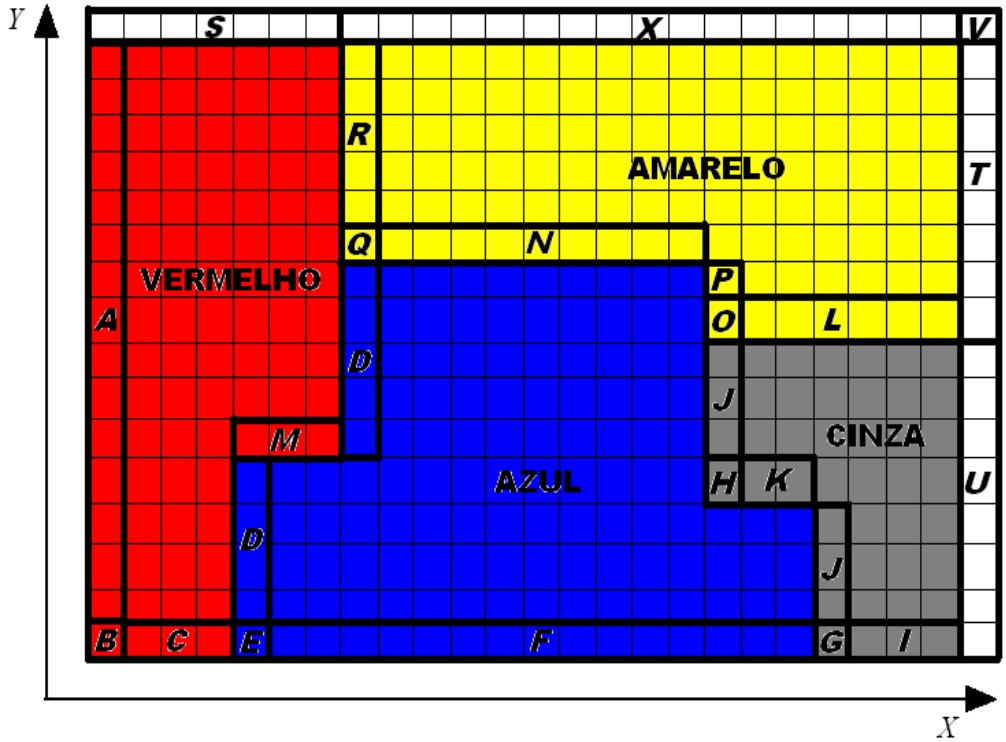

Figura 3.19. Divisão das células da malha em conjuntos de acordo com os coeficientes

3.7. A classe TProject

A classe TProject é a principal classe do programa. Nela se processa as informações vindas das outras classes e se calcula os campos elétrico e magnético das células da malha por meio do método das diferenças finitas no domínio do tempo. Ela está mostrada na figura 3.20 juntamente com alguns de seus métodos e atributos. Os atributos Ex, Ey, Ez, Hx, Hy, Hz são utilizados para armazenar os campos elétrico e magnético da malha nos intantes determinados. Os atributos Grid e Mesh são referências (ponteiros) a objetos das classes Tgrid e Tmesh, instaciados pela função membro New() de Tproject. O atributo LookUpTable é apenas utilizado como cópia deste mesmo atributo em Mesh. Isso é feito, apenas, para melhorar o desempenho durante o loop de processamento, já que ele poderá ser acessado diretamente e não por meio de uma chamada de função a mais à Mesh.

O atributo Objects é um vetor de ponteiros a objetos do tipo TFDTDObject. Tudo que é utilizado durante o processamento básico do programa é armazenado nesse vetor, desde as referências Grid e Mesh, até as fontes e os objetos tridimensionais presentes na malha computacional. Este atributo é fundamental, pois é por meio dele que o processamento polimórfico dos objetos é feito. A atualização dos campos elétrico e magnético é feita a partir do método Calculate(). Este método, porém, não realiza os cálculos diretamente, utiliza, para isso, funções adicionais dentro de um arquivo DLL (*Dynamic Link Library*) que realiza as contas necessárias. Este procedimento foi adotado apenas para melhorar a velocidade do programa, porém, todo os cálculos poderiam ser realizados dentro do próprio código fonte de TProject sem alteração alguma nos resultados.

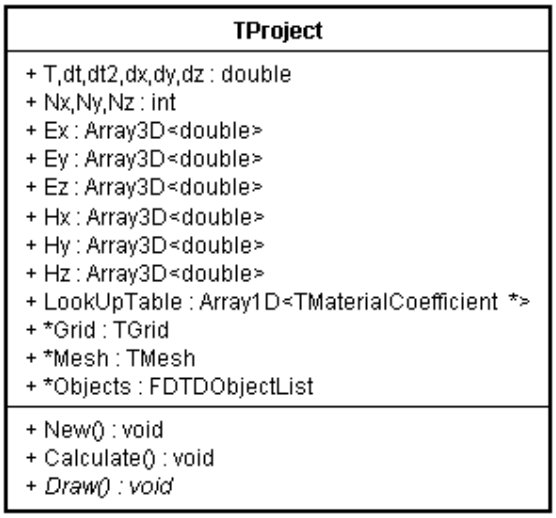

Figura 3.20. Classe Tproject

A seguir, serão descritos os métodos New() e Calculate() da classe TProject.

A figura 3.21 ilustra de maneira simplificada o método New() da classe Tproject. Na linha 3, inicializa-se um objeto TGrid e atribui o ponteiro resultante a Grid. Na linha 4, adiciona-se o ponteiro Grid ao vetor Objects. Na linha 5, igualmente, inicializa-se um objeto TMesh e atribui o ponteiro resultante a Mesh. Na linha 6, atribui-se ao ponteiro Grid de Mesh, o ponteiro Grid de TProject. Na linha 7, chama-se o método Initialize do objeto referenciado por Mesh. Este método, basicamente, atribui o número de células em cada dimensão grid aos inteiros Nx, Ny e Nz de TMesh. Finalmente, na linha 8, adiciona-se ao vetor Objects o ponteiro Mesh.

```
1   void TProject::New()
2   {
3     Grid = new TGrid(this);
4     Objects->Add(Grid);
5     Mesh = new TMesh(this);
6     Mesh->Grid=Grid;
7     Mesh->Initialize();
8     Objects->Add(Mesh);
9
10  }
```

Figura 3.21. Método New()

A figura 3.22 mostra de forma simplificada o método Calculate() da classe TProject. Na linha 3 é chamada a função Generate() de Mesh, onde é implementado o método do cruzamento dos raios para que seja determinado o vetor tridimensional Mat de Mesh. Entre as linhas 5 à 7 são atribuídos aos inteiros Nx, Ny e Nz o número de células de cada dimensão de Grid. Entre as linhas 9 e 11, os vetores tridimensionais dos campos elétrico e magnético, Ex, Ey, Ez, Hx, Hy e Hz, são inicializados atribuindo as dimensões adequadas e o valor zero a todos os elementos. Nas linhas 16 e 17 são inicializados, a partir dos atributos de Grid, os valores do intervalo de tempo, dt, e da metade do intervalo de tempo, dt2. Este é utilizado como incremento de tempo da variável T no loop de atualização dos campos, como propõe o método das diferenças finitas. Na linha 19 é chamado o método GetCoefficients() de Mesh. Como explicado anteriormente, esse método inicializa os vetores LookUpTable e Coefficiets de Mesh para que os coeficientes *Ca*, *Cb*, *Da* e *Db* das células da malha sejam conhecidos. Entre as linhas 20 e 24 ocorre a inicialização do vetor LookUpTable a partir da cópia deste mesmo parâmetro existente em Mesh. Na linha 25 é atribuído o valor zero a variável T. Esta variável apenas armazena o

valor do instante de tempo em que se está realizando o cálculo das componentes de campo. Entre as linhas 26 e 29 é passada para a DLL os parâmetros necessários para a atualização dos campos. Pode-se notar que os campos elétrico e magnético são passados por referência, assim, as modificações dos campos ocorridas dentro da DLL serão feitas também nos campos próprios de TProject. No código fonte da DLL, nota-se, também, que os campos são atribuídos a outros ponteiros, assim, mudando-se os valores dos campos fora da DLL, muda-se também o valor dos campos apontados pelos ponteiros dentro dela. Essa característica será fundamental, pois o cálculo dos campos pelos objetos será realizado fora da DLL. Entre as linhas 36 e 56 é feita a implementação do método das diferenças finitas no domínio do tempo. Na linha 36 é definida a instrução de repetição *for* que se repete tantas vezes quanto for o número de intervalos de tempo definido em Grid. Na linha 39, chama-se a função CalculateAllEFields() implementada na DLL. Ela calcula todos os campos elétricos das células da malha tridimensional. A instrução *for*, entre as linhas 41 e 44, percorre todo o vetor Objects e invoca a função *CalculateEFields()* para cada um de seus elementos. Observa-se aí o processamento polimórfico em que a função *virtual CalculateEFields()* dos objetos filhos é chamada a partir de um ponteiro da classe básica. Se essa função não fosse definida *virtual* na classe mãe, TFDTDObject, o processamento seria inadequado pois se invocaria a função definida nesta classe e não nas classes herdadas. Na linha 46, incrementa-se a variável T em dt/2, representando o instante em que serão calculados os campos magnéticos da malha. Na linha 48, invoca-se a função CalculateAllHFields() da DLL, que calcula todos os campos magnéticos do domínio. Entre as linhas 50 e 53, novamente é utilizado o polimorfismo para a chamada da função *virtual CalculateHFields()* de todos os elementos do vetor Objects. Na linha 55 incrementa-se novamente a variável T para então repetir o loop. Entre as linha 58 e 61, é realizado o pós-processamento de todos os objetos a partir da chamada da função *virtual PostProcess()* de todos os elementos de Objects, terminando-se, assim, o método Calculate().

```
1     void TProject::Calculate()
2  {
3     Mesh->Generate();
4
5     Nx=Grid->NumX;
6     Ny=Grid->NumY;
7     Nz=Grid->NumZ;
8
9     Ex= Array3D<double>(Nx,Ny+1,Nz+1,0.0);
10    Ey= Array3D<double>(Nx+1,Ny,Nz+1,0.0);
11    Ez= Array3D<double>(Nx+1,Ny+1,Nz,0.0);
12    Hx= Array3D<double>(Nx+1,Ny,Nz,0.0);
13    Hy= Array3D<double>(Nx,Ny+1,Nz,0.0);
14    Hz= Array3D<double>(Nx,Ny,Nz+1,0.0);
15
16    dt=Grid->dt;
17    dt2=Grid->dt/2;
18
19    Mesh->GetCoefficients();
20    LookUpTable= Array1D<TMaterialCoefficient *>(Mesh->LookUpTable.size());
21    for (int i=0; i < Mesh->LookUpTable.size(); i++)
22    {
23       LookUpTable[i]=Mesh->LookUpTable[i];
24    }
25    T=0;
26    SetFieldArrays(dt,Nx,Ny,Nz,&Grid->dx,&Grid->dy,&Grid->dz,
27                   &Grid->GridX,&Grid->GridY,&Grid->GridZ,
28                   &Ex,&Ey,&Ez,
29                   &Hx,&Hy,&Hz);
30    for (int ob=0;ob<Objects->size();ob++)
31    {
32        Objects->at(ob)->Init();
33    }
34    SetCoefficientArrays(&Mesh->Coefficients,LookUpTable.dim(),&LookUpTable);
35    // Implementação do Método das Diferenças Finitas:
36    for (t=0;t<Grid->TimeSteps;t++)
37    {
38       // Calculo dos campos E
39           CalculateAllEFields();
40     ////////////////////////////////////////////////
41       for (int ob=0;ob<Objects->size();ob++)
42       {
43           Objects->at(ob)->CalculateEFields();
44       }
45     ////////////////////////////////////////////////
46       T+=dt2;
47       // Calculo dos campos H
48          CalculateAllHFields();
49     ////////////////////////////////////////////////
50          for (int ob=0;ob<Objects->size();ob++)
51       {
52           Objects->at(ob)->CalculateHFields();
53       }
54     ////////////////////////////////////////////////
55       T+=dt2;
56    }
57    ////////////////////////////////////////////////
58    for (int ob=0;ob<Objects->size();ob++)
59       {
60           Objects->at(ob)->PostProcess();
61       }
62    ////////////////////////////////////////////////
63 }
```

Figura 3.22. Método Calculate()

Pode-se chegar, portanto, à figura 3.23, que mostra uma estrutura inicial da organização das classes no programa.

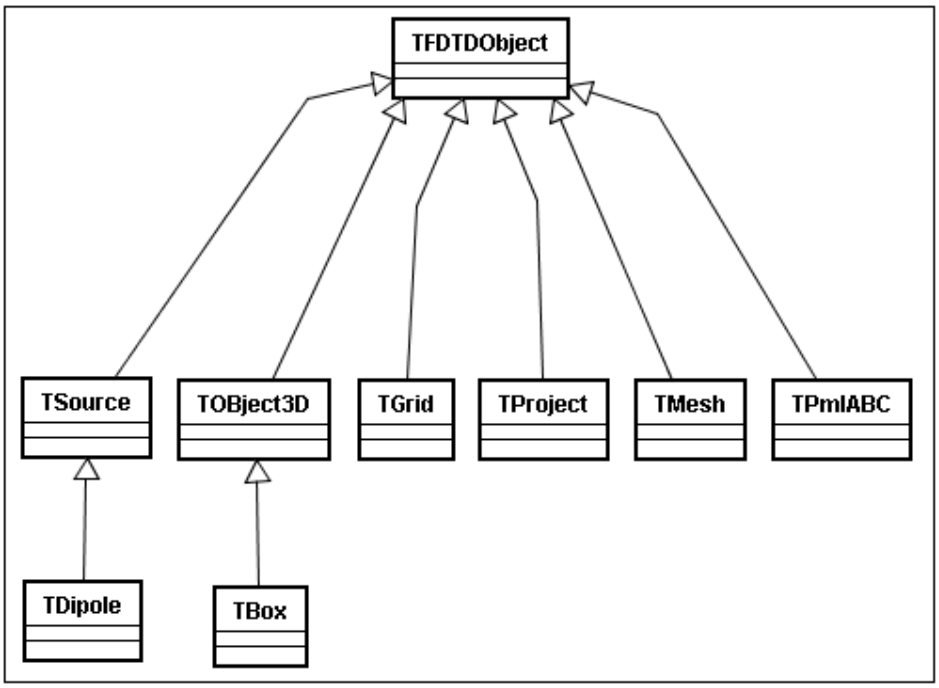

Figura 3.23. Estruturação inicial das classes do programa

3.8. A classe FDTDMain

A ligação entre a interface gráfica do FDTD Studio e as classes internas do programa se dá pela classe FDTDMain. Esta classe é bastante completa e centraliza tudo que é executado no programa, seja a iniciação de um novo projeto, a rotação dos objetos na malha computacional, a adição de um novo objeto, a abertura de um outro projeto, o salvamento do projeto atual, o início da simulação etc. Ela possui um acesso direto às classes internas do programa. A figura 3.24 ilustra esta classe com alguns de seus métodos e atributos. Como pode ser observada, essa classe possui como atributo um ponteiro Project que aponta para o objeto da classe TProject em execução. Os métodos InsertBoxExecute(), ActionPMLExecute(), ActionDipoleExecute() criam, respectivamente, ponteiros das classes TBox, TPmlABC e TDipole, e os adicionam à Project, inserindo os respectivos elementos ao seu vetor ObjectList. O método RunFDTDExecute() basicamente invoca a função Calculate() de Project.

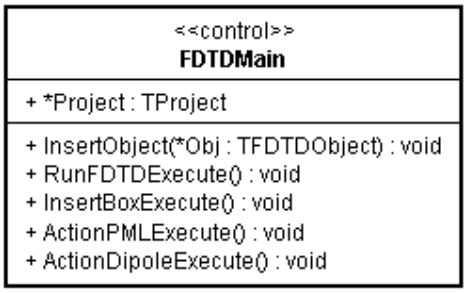

Figura 3.24. Classe FDTDMain

No Apêndice A, encontra-se um diagrama de classes mais completo que mostra os principais elementos para o processamento do programa e as relações existentes entre eles. Também são mostradas as classes isoladas com seus principais métodos e atributos, além de alguns diagramas de sequência importantes.

4. PROJETO DO SOFTWARE

4.1. Diagrama de casos de uso do FDTD Studio

A figura 4.1 mostra diagrama de casos de uso do programa, que, como explanado no capítulo 2, costuma ser utilizado no início da modelagem do sistema e é fundamental para o desenvolvimento dos demais diagramas representativos do software. A sua documentação está descrita no Apêndice B. Ele apresenta uma visão externa geral das funções e serviços que o sistema deverá oferecer aos usuários e deve sempre ser consultado durante o desenvolvimento. Porém, não pode ser visto como algo fixo, já que deve ir se adequando, sempre, ao projeto do software. Este diagrama e sua documentação foram construídos analisando o próprio FDTD Studio e a referência [1].

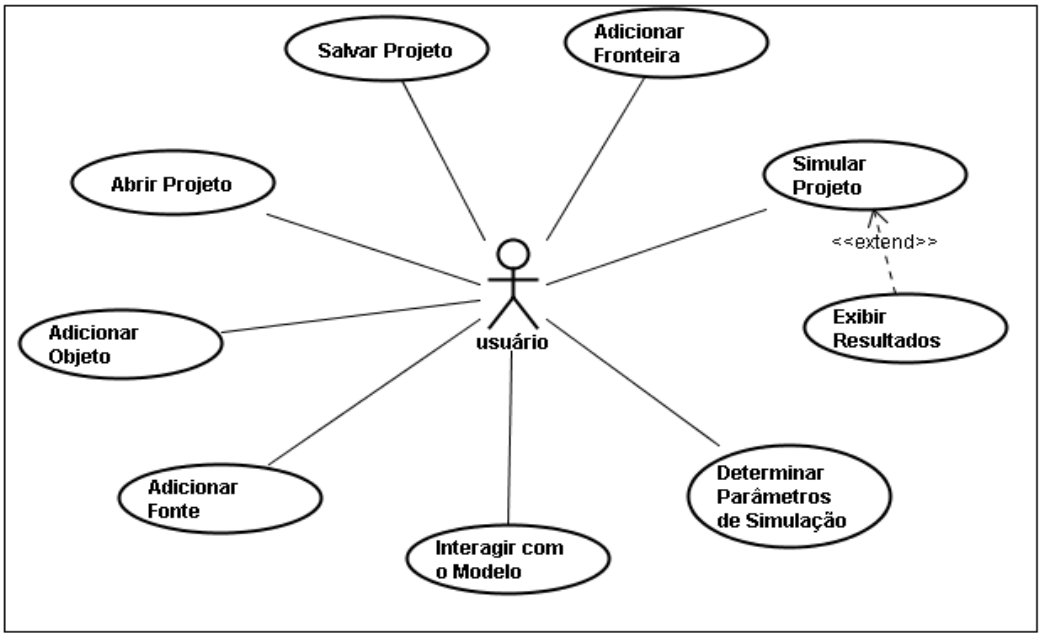

Figura 4.1. Diagrama de casos de uso do projeto

4.2. Organização inicial dos componentes do software

A mesma ideia usada no software Gmat, descrito na referência [8], será empregada para organizar os componentes do sistema em quatro pacotes básicos. Esses pacotes, mostrados na figura 4.2, não significam, necessariamente, uma organização física dos

componentes, mas sim, uma organização em termos de funcionalidade. Eles devem interagir entre si durante a execução do programa.

Como pode ser visto, os elementos do FDTD Studio podem ser divididos em quatro pacotes, denominados *Model*, *Engine*, *Program Interfaces* e *Utilities*. Dentro de cada pacote, serão agrupados muitos dos componentes já existentes no programa original e outros que serão criados para a elaboração do novo programa. Cada pocote descrito pode ser novamente quebrado em sub-pacotes, como mostra a figura. Dentro desses sub-pacotes devem estar presentes as classes do programa propriamente ditas.

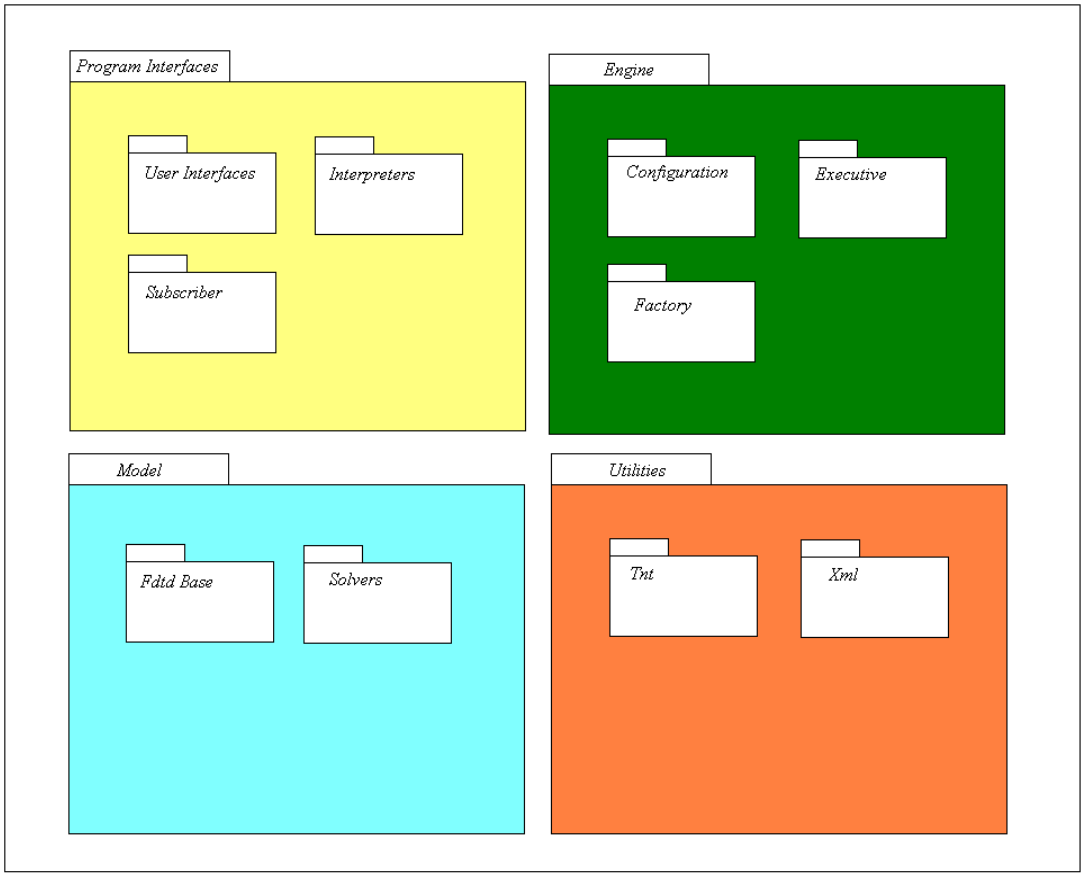

Figura 4.2. Organização em pacotes dos componentes do software

4.3. Descrição dos Pacotes e dos Sub-pacotes

O pacote *Program Interfaces* é responsável por qualquer forma de interação entre os usuários externos ao programa e o FDTD Studio. Ele pode ser dividido em três subpacotes:

- *User Interface*: este sub-pacote deve conter todos os elementos gráficos básicos do programa tais como botões, janelas, barras de ferrramentas etc. Esses elementos devem ser elaborados utilizando a biblioteca wxWidgets, que é livre e pode ser utlizada em diversas plataformas. Também é nesse pacote que deve ser elaborado a visualização gráfica da malha computacional do software utilizando a linguagem OpenGL.
- *Interpreters*: este sub-pacote deve conter os componentes responsáveis pela ligação entre os componentes de User Interface e os elementos internos do programa. Ela deve traduzir as ações dos usuários em métodos adequados a serem chamados em outras classes, sobretudo, na classe TModerator que será vista a seguir.
- *Subscribers*: este pacote deve conter os componentes responsáveis pela visualização gráfica dos resultados.

O pacote *Engine* possui componentes responsáveis pelo controle de todos os outros objetos do sistema. Ele recebe instruçoes diretamente da interface. Pode ser dividido, também, em três subpacotes:

- *Executive*: este sub-pacote contém o componente de processamento central do FDTD Studio denominado TModerator. Tal componente recebe as mensagens da interface pelos componentes de *Interpreters* e determina qual ação dever ser tomada. A classe TProject também faz parte de *Executive*, que deve conter, também, a classe TPublisher. Esta tem por função distribuir os resultados da simulação aos componentes do sub-pacote *Subscribers*, referente à interface.
- *Configuration*: este sub-pacote contém um componente denominado TConfigManager, que é utilizado para armazenar todos objetos instanciados que comporão a classe FDTDObjectList.
- *Factory*: este sub-pacote contém um componente denominado TFactoryManager, que é responsável pela criação de todos os objetos que serão armazenados em FDTDObjectList, além da criação do solver, explicado posteriormente. Se o usuário quiser acoplar algum outro tipo de objeto no software, basta adicionar o método de instanciação correspondente no TFactoryManager. A existência deste componente, com a função exclusiva de criar objetos, torna o software mais extensível.

O pacote *Model* é onde se localizam todos os elementos utilizados pelos componentes do pacote *Engine*. É nele que estão localizados todas as classes que podem compor o vetor FDTDObjectList, a classe Solver, além de diversas classes úteis que

podem ser utilizadas como atributos de outras classes. Pode ser dividido em dois subpacotes:

- *Fdtd Base*: este sub-pacote contém todas as classes básicas do projeto, desde aquelas derivadas da classe TFDTDObject, a classes úteis que podem ser utilizadas como atributos de outras, tais como TVertex e TFacet. Todas as classes presentes no diagrama de classes do Apêndice A compõem este sub-pacote, com excessão das classes Tproject e FDTDObjectList, que fazem parte do pacote *Engine*.
- *Solver*: este sub-pacote contém um componente de mesmo nome, Solver. Ele foi criado para que o processo de cálculos das componentes de campo fosse desacoplado da classe TProject. Com isso, consegue se obter uma maior modularidade dos componentes do sistema.

O pacote *Utilities* contém componentes auxiliares que são utilizados para facilitar o processamento do programa. Faz parte desse pacote, por exemplo, a biblioteca *TNT*, que é utilizada para manipulação de arrays.

4.4. Interação entre os pacotes do sistema

A figura 4.3 mostra como alguns pacotes destacados, em **negrito**, interagem entre si e com outros componentes do sistema, em *itálico*. Os elementos estão inseridos em caixas com cores iguais aos dos respectivos pacotes. As setas duplas indicam um duplo sentido no fluxo de dados e informações. Os usuários interagem com o software por meio dos componentes do *User Interface*, que são construídos utilizando a biblioteca wxWidgets. Esses componentes, por sua vez, se comunicam com o pacote *Engine* através dos elementos do pacote *Interpreters*.

O único componente do Engine que se relaciona diretamente com o pacote Interpreters é a classe TModerator, que possui uma instanciação única durante toda execução do programa. Utilizou-se, para isso, o padrão de criação GoF, *Singleton*. Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto. Além do TModerator, TProject, TFactoryManager e TConfigManager também foram elaborados utilizando *Singleton.*

O TModerator, como dito anteriormente, é o componente central de processamento do software. As informações vindas da interface são processadas por ele, que, assim,

dispara os métodos necessários em diversas outras classes. Além disso, ele controla todo o fluxo de dados entre os diversos pacotes do programa.

As intruções básicas que podem ser requeridas pelo usuário durante a execução do programa são a criação de um novo objeto a ser inserido na malha computacional, a modificacão de algum objeto criado, a retirada de um objeto da malha computacional e a simulação do projeto. A interface básica do software assim como as classes TModerator, TFactoryManager e TConfigManger devem ser primordialmente elaboradas para que essas intruçoes sejam realizadas satisfatoriamente. Estas classes, por sinal, foram as principais novas classes criadas para que o software antigo tivesse uma melhor estrutura.

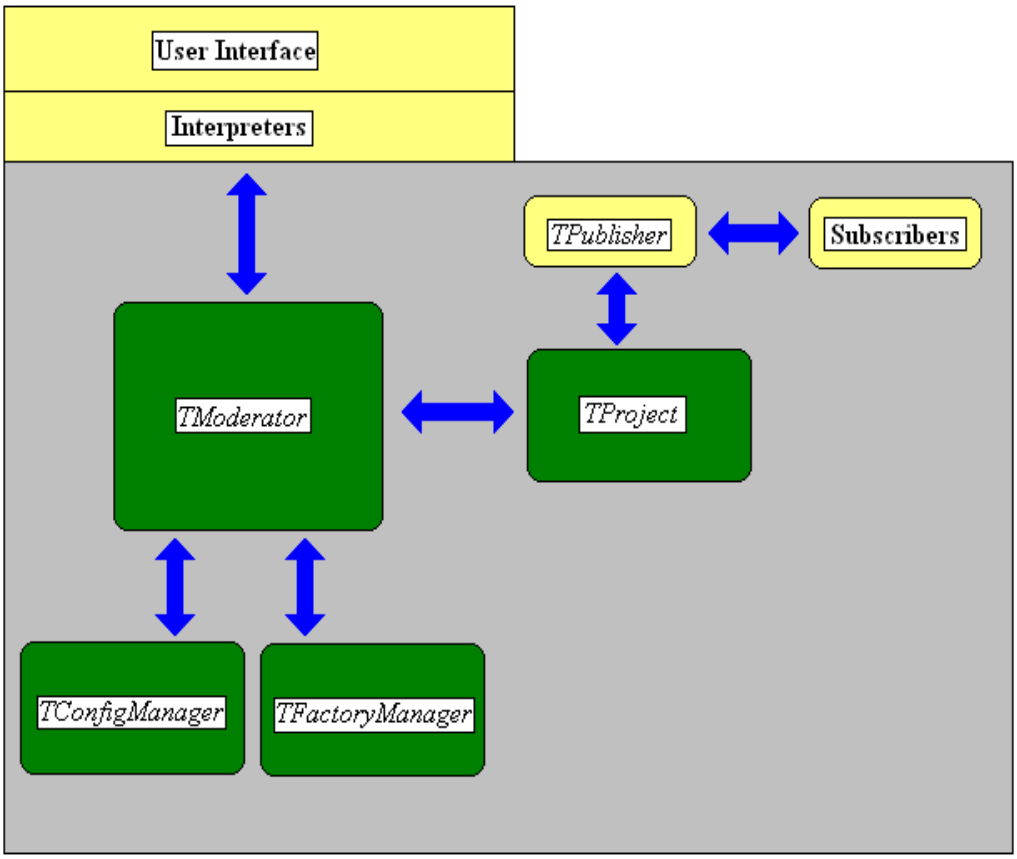

Figura 4.3. Interação entre os pacotes e componentes do sistema

Na criação, por exemplo, de um objeto a ser inserido na malha computacional, pode-se descrever uma séria de intruções a ser executadas pelos elementos da figura 4.3. Primeiramente, uma mensagem, vinda da interfade, instrui o TModerator a criar o objeto desejado. Este, por sua vez, solicita, ao TFactoryManager, a criação do objeto. O TFactoryManager cria o objeto requerido e retorna o seu ponteiro à TModerator. Por fim,

o TModerator encaminha o ponteiro, vindo do TFactoryManager, ao TConfigManager, que o armazena.

O TConfigManager sempre será acessado para que os objetos criados possam ser modificados, retirados, ou encaminhados à TProject para que a simulação seja executada. Neste último caso, é realizado uma cópia dos objetos armazenados no TConfigManager para TProject.

Ao ser requerida a exibição dos resultados da simulação, o TModerator deve solicitar isso ao TProject, que, por sua vez, instrui aos objetos da malha computacional que enviem ao TPublisher os dados necessários. Este solicita a exibição dos resultados aos componentes do *Subscribers.*

À seguir, serão descritas as classes TModerator, TFactoryManager e TConfigManager, além do funcionamento de diversos aspectos do software através de uma série de diagramas e figuras. Será focado o software no atual estágio de desenvolvimento. Portanto, os diagramas mostrados deverão ser revistos com o tempo com a inclusão de novos métodos e classes. A estrutura inicial da interface, por exemplo, ainda está bastante incipiente na nova configuração do programa. Além disso, algumas classes ainda não foram criadas e falta incluir estruturas de tratamento de exceções no código.

4.5. A classe TModerator

A classe TModerator é o ponto de entrada para os componentes do pacote *Engine*. Ele controla o fluxo de dados no programa, ordenando a criação de componentes no TFactoryManager, armazenando-os no TConfigManager e utilizando-os posteriormente para montar o TProject. TModerator também é responsável pela criação dos próprios componentes do pacote *Engine*. Ele funciona como um meio de acesso às classes internas do programa pelos usuários. Ele foi elaborado utilizando-se, além do padrão de projeto *Singleton*, anteriormente descrito, o padrão GoF *Mediator*, que é do tipo comportamental. Este padrão centraliza a comunicação entre objetos em um único elemento, o que simplifica as interfaces que os objetos precisam suportar e favorece a desacoplação entre eles.

Pela figura 4.3, pode-se observar o papel central que o componente TModerator tem na comunicação entre os vários elementos do *Engine* e do *Program Interfaces*. Essa estrutura é fundamental para diminuir a complexidade que se teria ao realizar uma comunicação direta entre os elementos. Essa complexidade seria, sobretudo, nas interfaces dos componentes, que teriam que possuir uma grande capacidade de realizar os mais diversos tipos de comunicações diferentes. O TModerator, portanto, centraliza toda essa complexidade em um único elemento, permitindo que as interfaces dos outros componentes sejam reduzidas ao mínimo necessário para realizar suas funções. Tudo isso torna os componentes do Engine fáceis de serem compreendidos e mantidos.

A figura 4.4 ilustra a classe TModerator com alguns de seus principais métodos e atributos. Ela possui três atributos fundamentais que são ponteiros aos outros componentes do *Engine*: Tproject, TConfigManager e TFactoryManager. O método Initialize() deve ser chamado logo após a criação da classe TModerator. Ele é utilizado para instanciar os outros elementos do *Engine*, além de configurar adequadamente TProject com os atributos necessários. Os métodos AddGridToProject(), AddMeshToProject() e AddSolverToProject() são utilizados para adicionar os elementos TGrid, TMesh e TSover à TProject. Esses elementos, essenciais para TProject, são adicionados dentro do método Initialize(). O métodoAddObjectsToProject () é chamado antes do início da simulação e é utilizado para copiar os objetos armazenados no TConfigManager à TProject. Os métodos CreateGrid(), CreateMesh(), CreateSolver(), CreateDipole(), CreateBox() e CreatePml() são utilizados, respectivamente, para criar instanciações de objetos do tipo TGrid, TMesh, TSolver, TDipole, TBox e TPmlABC. Os três primeiros são chamados dentro do método Initialize(), enquanto que os outros devem ser chamados de acordo com a necessidade do usuário. O método RunFDTD() inicializa o início da simulação propriamente dita.

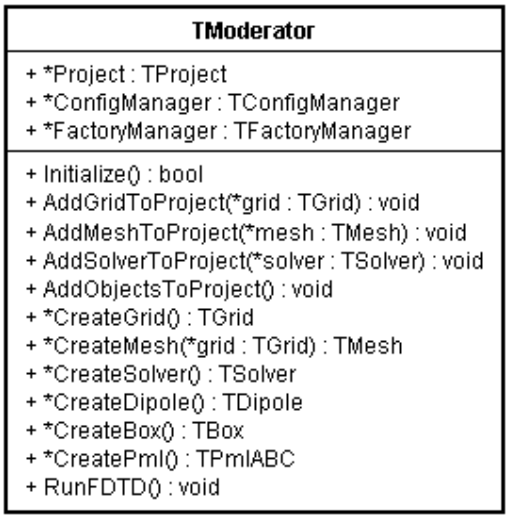

Figura 4.4. Classe TModerator

4.6. A classe TFactoryManager

A classe TFactoryManager é utilizada pelo programa na criação de todos os objetos do pacote *Model*. Ela também é estruturada pelo padrão Singleton, ou seja, existe apenas uma instanciação da classe TFactoryManager durante toda execução do software. Com ela, a adição de novos componentes ao sistema se torna mais simples e fácil de manter. A figura 4.5 ilustra essa classe e seus principais métodos. Os métodos CreateGrid(), CreateMesh(), CreateSolver(), CreateDipole(), CreateBox() e CreatePml() são utilizados para instanciar objetos das classes TGrid, TMesh, TDipole, TBox e TPmlABC, respectivamente, e retornam os ponteiros referentes aos objetos criados. Todos esses métodos são chamados pelas funções correspondentes em TModerator. Como esperado, TModerator acessa TFactoryManager para que este crie os objetos requeridos.

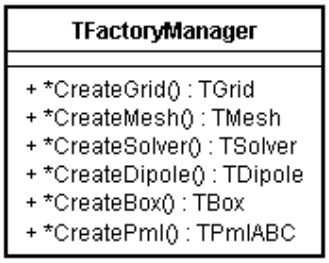

Figura 4.5. Classe TFactoryManager

4.7. A classe TConfigManager

A classe TConfigManager é responsável pelo armazenamento dos objetos criados em TFactoryManager. Ela, também, é criada utilizando o padrão Singleton. A figura 4.6 ilustra essa classe e seus principais métodos e atributos. O seu atributo principal é Objects, um ponteiro que aponta para o vetor FDTDObjectList, que é utilizado para armazenar todos os objetos que compõem a malha computacional. O método getObjects() retorna o ponteiro Objects e é chamado por TModerator antes do início da simulação. Por fim, o método AddObject() é utilizado para armazenar os objetos criados por TFactoryManager no vetor apontado por Objects. Ele, também, é chamado por TModerator.

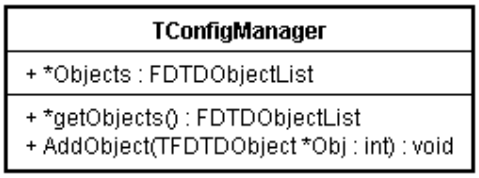

Figura 4.6. Classe TconfigManager

4.8. Diagramas de sequência importantes

Nesta seção, por meio de diagramas de sequência, serão descritos alguns processos importantes do FDTDStudio. Com eles será possível entender de maneira mais detalhada como ocorre o relacionamento entre as classes do pacote *Engine*, quais sequências de eventos são chamadas, e quais métodos são disparados entre os objetos envolvidos. Os componentes da interface do sistema não estão ainda plenamente desenvolvidos e foram representados nos diagramas por um único elemento denominado Fronteira.

O diagrama de sequência da figura 4.7 mostra como ocorre o processo de inicialização do software e, mais especificadamente, as sequências de eventos disparadas pelo método Initialize() da classe TModerator. Primeiramente é chamado o método Instance() de TModerator. Este método retorna referência única à TModerator e é com ele que o padrão Singleton é implementado. O método Initialize() é então chamado. Ele é responsável por criar os componentes básicos do *Engine* e os que devem compor o projeto corrente. Inicialmente, este método instancia os elementos das classes TProject,

TFactoryManager e TConfig Manager, componentes do pacote *Engine*. Esses elementos são primordiais e devem existir antes que o usuário possa manipular os componentes do sistema. Como pode ser visto, eles também são criados a partir da função Instance() de cada um, de forma a implementar o padrão Singleton.

A seguir, nos blocos identificados por A e B são instanciados os componentes da classe TGrid e TSolver. No bloco A, primeiramente, é feita uma auto-chamada da função membro CreateGrid(). Esta, por sua vez, acessa a função correspondente em FactoryManager, que, efetivamente, cria o objeto solicitado. É chamado, então, a função AddObjects de FactoryManager. Ela armazena o Grid recém criado no vetor de objetos dessa classe. É feita, então, uma auto-chamada à função AddGridToProject, que adiciona o Grid criado à Project. No bloco B, inicialmente é feita uma auto-chamada da função CreateSolver, que, por sua vez, chama a função correspondente em FactoryManager. Este invoca a função Instance() de Solver, que também é único no programa. A seguir a função AddSolverToProject é chamada para adicionar o Solver ao projeto corrente. Nota-se, no bloco B, que o Solver não é adicionado à ConfigManager, já que não faz parte da malha computacional propriamente dita. Faltou incluir no diagrama um bloco C, que instancia um elemento TMesh. Este bloco possui as mesmas funções do A, porém, para TMesh. Ele não foi incluído para que o diagrama não ficasse demais sobrecarregado.

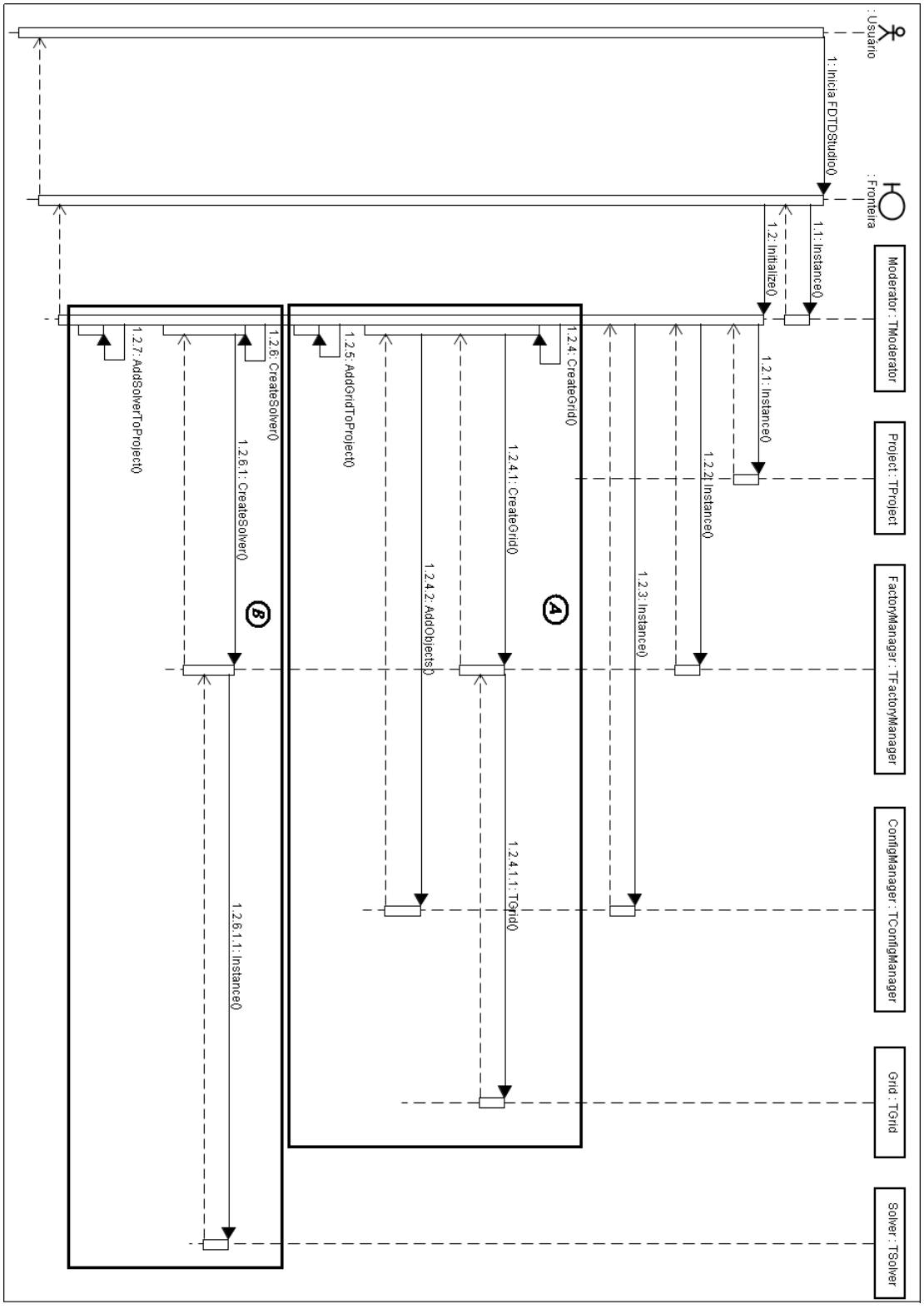

Figura 4.7. Diagrama de sequência de inicialização do software

O diagrama de sequência da figura 4.8 ilustra o processo que ocorre na solicitação pelo usuário da adição de uma antena dipolo à malha computacional. Ao ser solicitada, a fronteira invoca o método CreateDipole() de Moderator. Este método chama a função de mesmo nome em FactoryManager que, efetivamente, cria o objeto da classe TDipole solicitado. Novamente, em Moderator, o método CreateDipole() invoca a função AddObject() de ConfigManager, que armazena o objeto recém criado ao vetor FDTDObjectList dessa classe. Pode-se notar que todo o fluxo de informações entre os elementos do *Engine* se dá pelo Moderator. Ele centraliza toda comunicação entre esses componentes.

Ao adicionar qualquer outro tipo de objeto à malha computacional, o diagrama de sequência correspondente é semelhante ao descrito na figura 4.8.

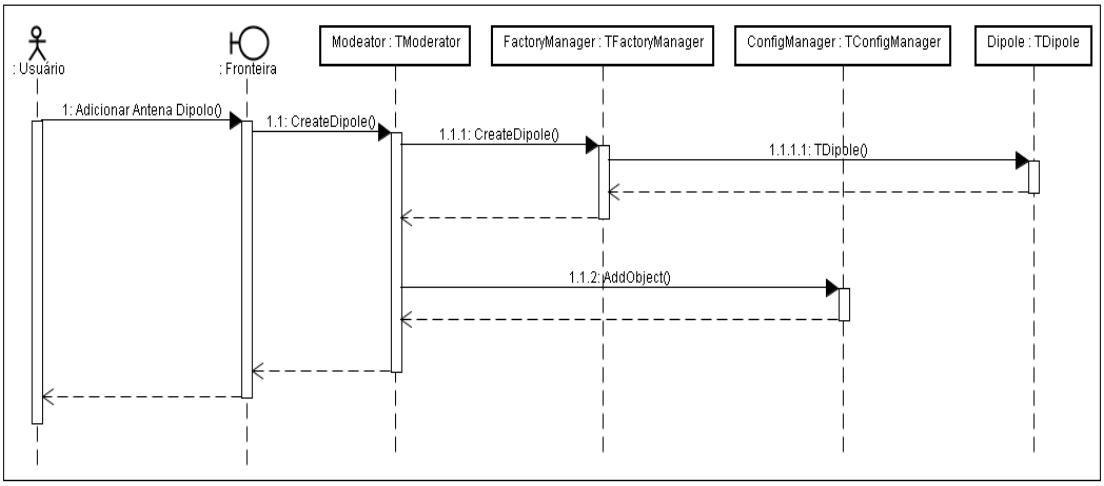

Figura 4.8. Diagrama de sequência de adicionar uma antena dipolo

O diagrama de sequência da figura 4.9 ilustra o processo que ocorre na simulação do projeto. Ao ser solicitada a simulação, a fronteira invoca o método RunFDTD de Moderator. Este realiza uma auto-chamada da função AddObjectsToProject, que, basicamente, copia, através da função GetObjects, o vetor de objetos de ConfigManager e o atribui ao parâmetro correspondente em Project. Novamente, em Moderator, o método RunFDTD chama a função Calculate() de Project que, por sua vez, invoca a função Calculate() de Solver. É neste elemento que os cálculos da simulação são efetivamente feitos, portanto os cálculos dos campos estão isolados em uma classe única, desacoplada de TProject.

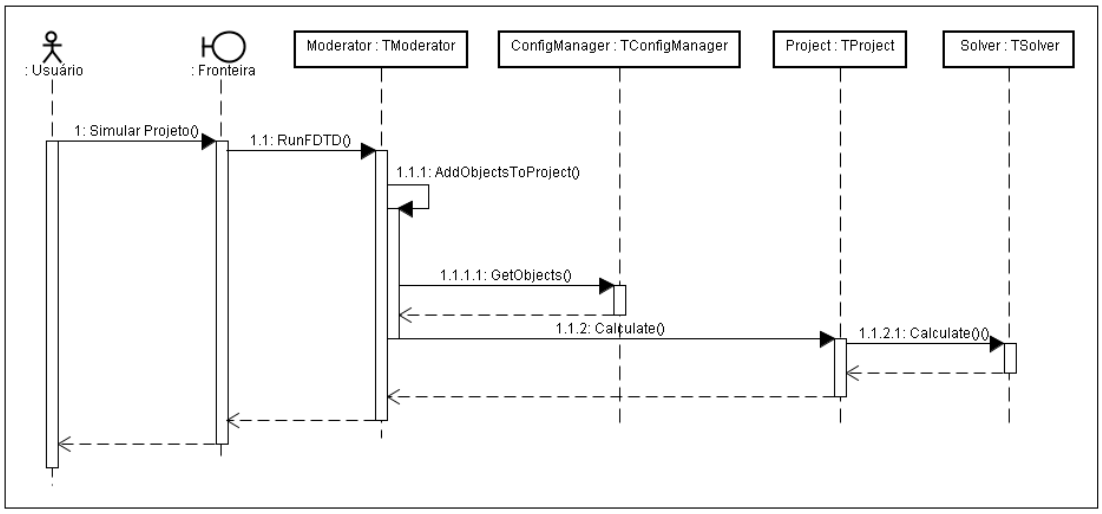

Figura 4.9. Diagrama de sequência da simulação do projeto

A figura 4.9 mostra a tela inicial do novo programa. Os componentes da interface ainda estão em fase de desenvolvimento e o protótipo mostrado não está na sua versão final. A interface com o usuário será semelhante a do programa original, possuindo uma área com os objetos disponíveis, uma com os objetos já presentes na malha, uma caixa de texto para exibir as informações pertinentes e uma área para exibir a visualização gráfica do domínio computacional utilizando OpenGl.

Figura 4.10. Tela inicial do novo programa

4.9. Aperfeiçoamento dos elementos do pacote *Model*

No Apêndice A, estão descritos os principais classes do pacote *Model*. As estruturas internas dessas classes, praticamente, não foram alteradas, com exceção da substituição de alguns elementos exclusivos da biblioteca *Borland*, por outros, da *C++ Standart Library*. Essas classes devem ser aperfeiçoadas com a utilização de classes básicas abstratas. O propósito de tais classes é fornecer uma classe básica apropriada a partir da qual outras classes podem herdar. Muitos bons sistemas orientados a objetos têm hierarquia de classes encabeçada por classes básicas abstratas [7]. A classe FDTDObject, por exemplo, deveria ser abstrata, já que ela não possui nenhuma instanciação direta. Deve-se assim adequar algumas das funções virtuais para que elas sejam virtuais puras, o que caracteriza as classes abstratas em C++.

Na Linguagem de Programação C++, existem três tipos de especificadores de acesso: *public*, *private* e *friend*. O especificador de acesso *public* indica que a função ou atributo de uma classe pode ser chamado por outras funções do programa e por funções-membros de outras classes. O especificador de acesso *private* indica que a função ou atributo apenas pode ser acessado por funções-membro da própria classe. O especificador de acesso *friend* é definido para funções fora do escopo da classe e permite que essas funções acessem membros *public* e *private* da classe. Esse especificador também pode ser atribuído a alguma classe, indicando que todas as suas funções são declaradas como amigas de outra classe.

Os atributos de todas as classes do pacote *Model* foram declarados com o especificador de acesso *public*, o que não está de acordo com as premissas de uma programação robusta e confiável. Tais atributos devem ser declarados com o especificador de acesso *private* em um processo conhecido como ocultação de dados [7]. Assim, os membros de dados das classes são encapsulados (ocultados) e poderiam ser acessados apenas por funções-membros das classes dos objetos, que devem ser declaradas *public*. Transformar os membros de dados de uma classe public e as funções-membros, private, facilita a depuração, pois os erros relacionados à manipulação de dados vão sendo encontrados nas funções-membro da classe e para os *friends* da classe [7].

Seria interessante, também, criar funções membros para *configurar* e para *obter* os membros de dados *private* das classes. Uma conveção comum é iniciar o nome de tais funções com *set*, para as funções que atribuem valores, modificadoras, e com *get*, para as

funções que obtêm os valores, funções de acesso [7]. Com isso se torna possível que os clientes de uma classe acessem seus dados ocultos, *private*, porém indiretamente. O cliente tem consciência que está tentando modificar ou acessar esses dados porem não sabe como a função realiza isso internamente. Inclusive as funções-membros próprias da classe devem acessar os seus dados utilizando essas funções auxiliares, o que torna a classe mais fácil de manter, mais expansível e com menos probabilidade de parar. Por exemplo, caso deseje alterar o nome e o tipo de dados a serem acessados, apenas as funções *get* e *set* sofreriam alterações internamente, mas não as funções que se utilizam dessas funções auxiliares.

5. Conclusões

Este trabalho estudou a analisou o programa FDTD Studio. Discorreu-se, também, sobre os principais recursos e ferramentas de desenvolvimento de software

Foram isoladas as principais classes do programa original e retirou-se a dependência que inúmeros de seus elementos tinham com bibliotecas restritas. Com isso, foi possível realizar algumas simulações diretamente a partir do código fonte, sem o auxílio da interface. Estudaram-se como os elementos se relacionam e como eles trabalham conjuntamente para a realização de simulações. Realizaram-se inúmeros diagramas e uma análise detalhada de trechos específicos de código.

Foram mostradas as principais características relacionadas à Engenharia de Software e suas aplicabilidades no desenvolvimento de programas. Foi mostrada a Linguagem de Modelagem Unificada e seus inúmeros recursos. Mostraram-se, também, as características principais da linguagem de programação C++ e suas vantagens. Analisaram-se alguns padrões de projeto e o processo de desenvolvimento de software iterativo e incremental.

Desenvolveu-se a estrutura inicial de um novo sistema que visa aperfeiçoar o FDTD Studio, sobretudo, na comunicação entre seus componentes e na interação com o usuário. Criaram-se componentes básicos como as classes TModerator, TFactoryManager e TConfigManager, que visam isolar determinados processos e centralizar as complexidades existentes em um único elemento. Isso trouxe uma maior modularidade ao programa melhorando a sua capacidade de expansão e manutenção. Foi frisado, inclusive, melhorias a serem feitas no código fonte atual de forma a trazer benefícios quanto à prevenção de erros e expansividade.

Espera-se que esse novo sistema e que a documentação aqui descrita sirva de ponto de partida para o aperfeiçoamento posterior do software. Deve-se desenvolve-lo, ao máximo, tornando-o mais robusto, confiável e com características de expansividade maximizadas. Um aspecto a ser focado é a interface com o usuário, que ainda está bastante incipiente e precisa ser melhorada com a criação das novas classes e com a implementação das funções necessárias nos elementos já existentes. Após essa fase, os outros objetos do programa original podem ser incorporados, adequando-os ao novo sistema e retirando as dependências com bibliotecas restritas.

Os diversos recursos de engenharia de software precisam ser analisados mais profundamente e aplicados de maneira intensa durante os novos desenvolvimentos. É

interessante, que o processo de desenvolvimento iterativo e incremental seja usado eficientemente. A aplicação do RUP, por exemplo, trará benefícios quanto ao direcionamento dos esforços em cada etapa do trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] PICANÇO, R. P. (2006). Desenvolvimento de uma Interface Integrada para o Projeto e Análise de Antenas Utilizando o Método das Diferenças Finitas no Domínio do Tempo (FDTD). Dissertação de mestrado. ENE/UnB, 2006.

[2] GUEDES, Gilleanes T. A. *UML:* uma abordagem pratica. São Paulo: Novatec

[3] http://pt.wikipedia.org/wiki/Desenvolvimento_iterativo_e_incremental

[4] http://www.wthreex.com/rup/

[5] LARMAN, Craig. *Utilizando UML e padrões*: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3ª. ed. Porto Alegre: Bookman

[6] STROUSTRUP, Bjarne. Linguagem de programacao c++(a). 3. ed. Porto alegre: Bookman.

[7] DEITEL, Harvey M.. C++: Como programar. 5ª. ed.

[8] http://gmat.gsfc.nasa.gov/downloads/documentation.html

[9] SOMMERVILLE, Ian. Engenharia *de software.* 8. ed. São Paulo: Pearson Addison Wesley

[10] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *UML*: guia do usuario. Rio de Janeiro: Campus, 2000.

[11] http://pt.wikipedia.org/wiki/RUP

APÊNDICE A

Na figura A.1 está mostrada as principais classes utilizadas para o processamento básico do software FDTD Studio. As classes TVertex estão destacadas em amarelo pois são as mesmas. Estão separadas para não sobrecarregar o desenho.

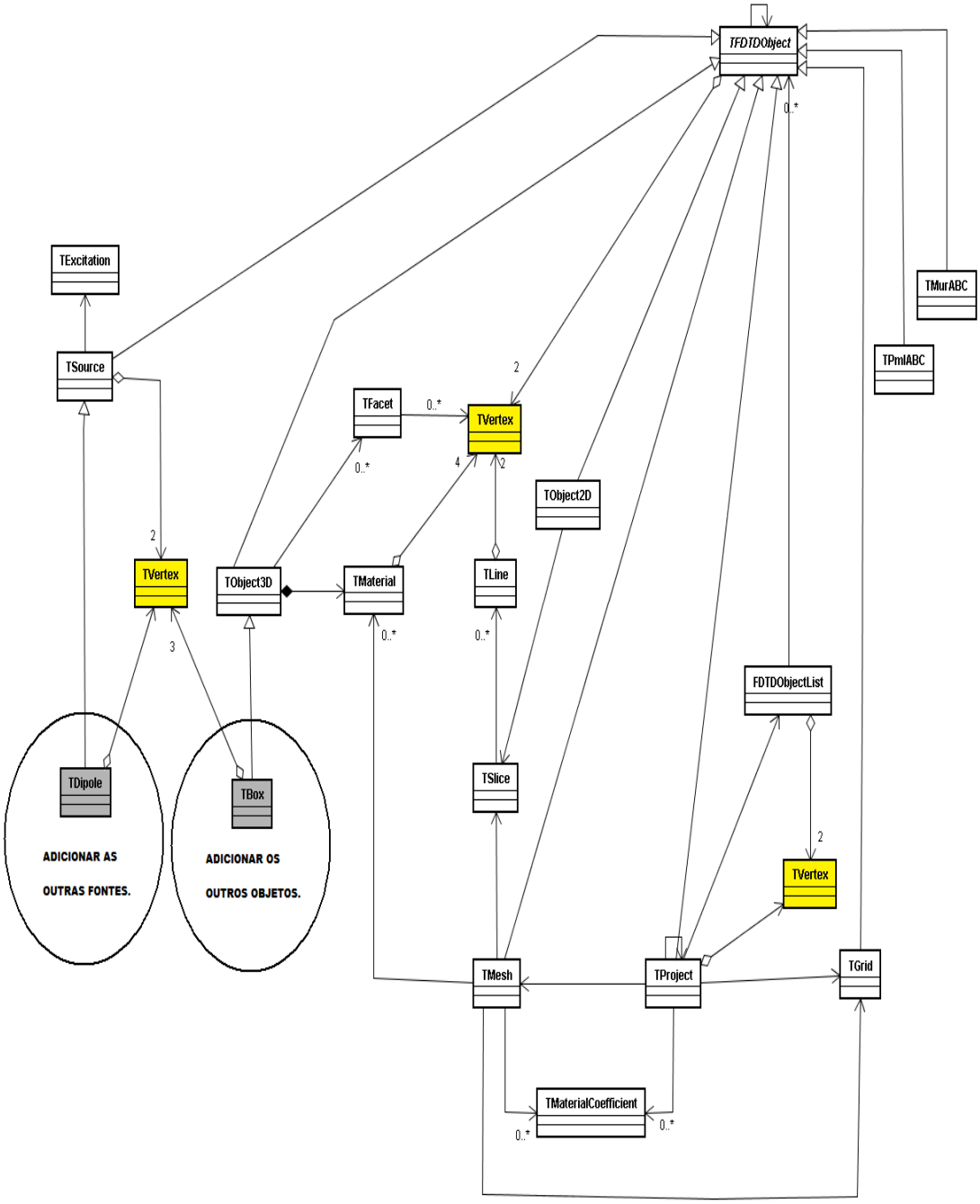

Figura A.1. Diagrama de classes básico do sistema

Nas figuras A.2 a A.20 estão mostradas estão mostradas a descrição básica das classes mostradas na figura A.1. Nas figuras de A.21 a A.25 estão mostrados diversos diagramas de sequência importantes do programa original.

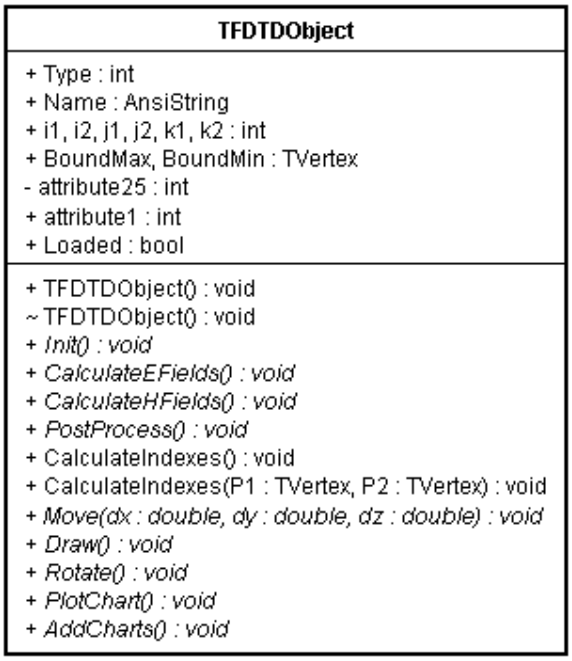

Figura A.2. Classe TFDTDObject

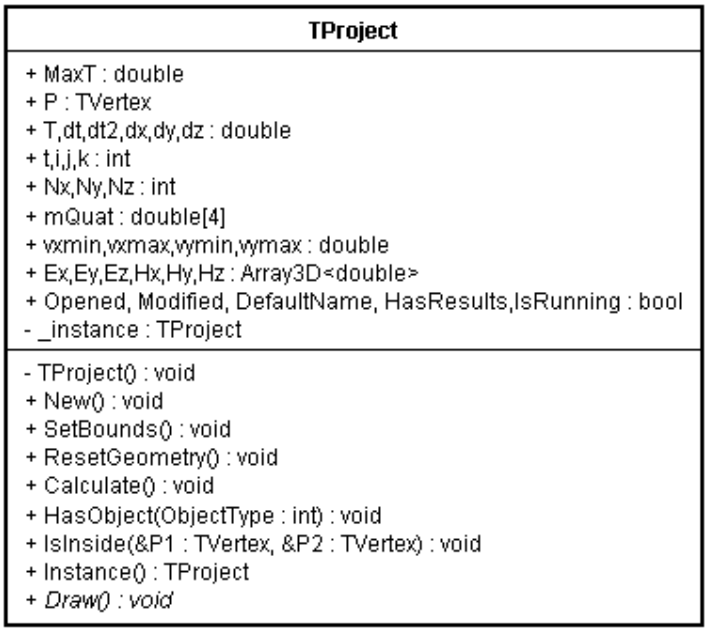

Figura A.3. Classe TProject

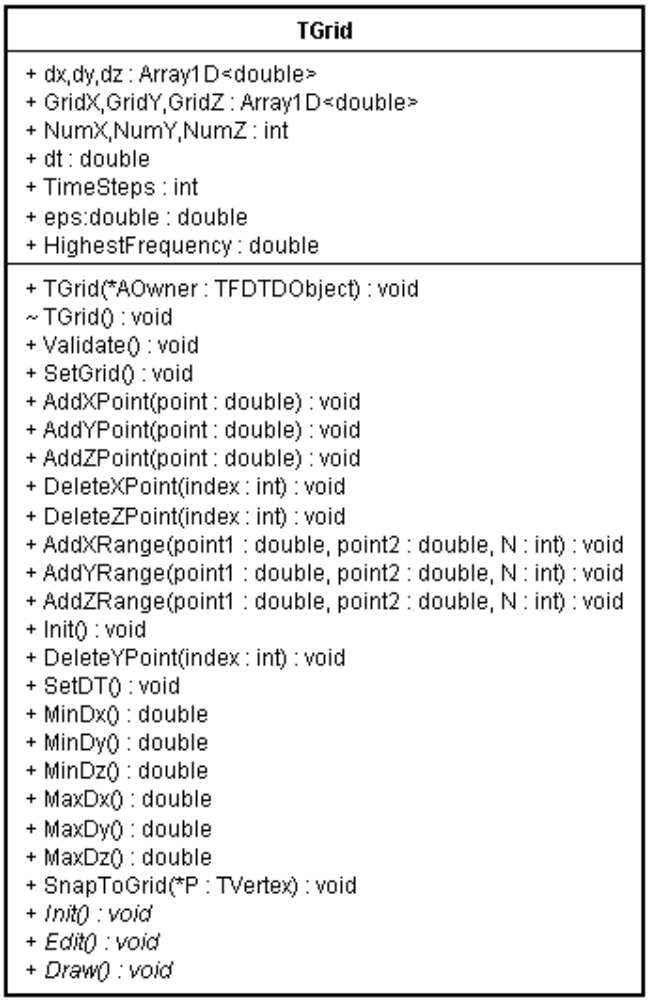

Figura A.4. Classe TGrid

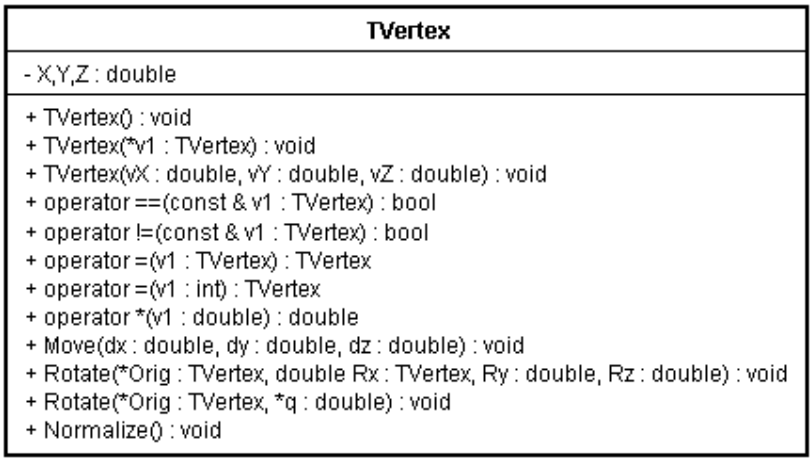

Figura A.5. Classe TVertex

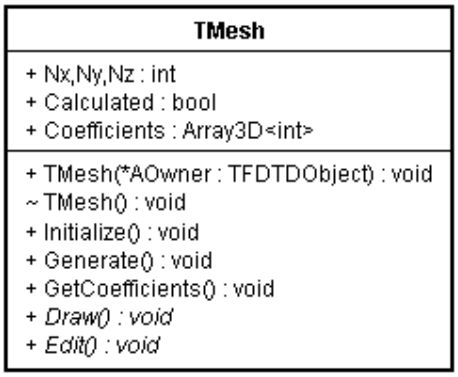

Figura A.6. Classe TMesh

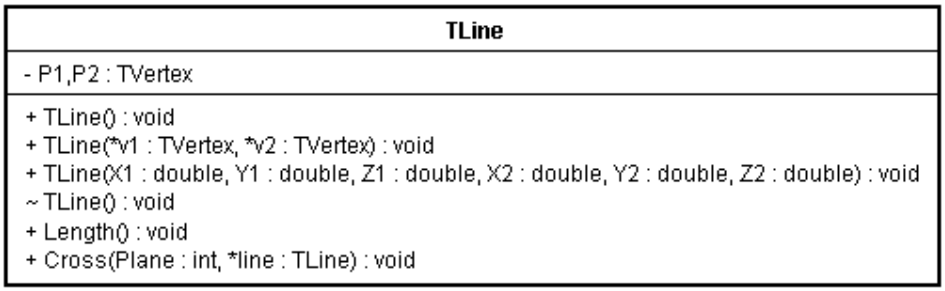

Figura A.7. Classe TLine

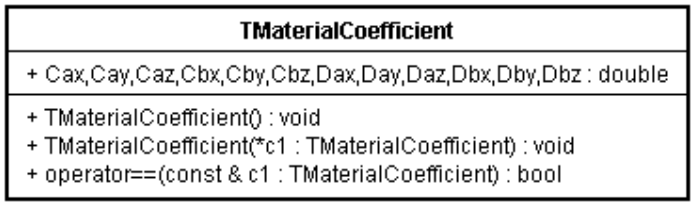

Figura A.8. Classe TMaterialCoefficient

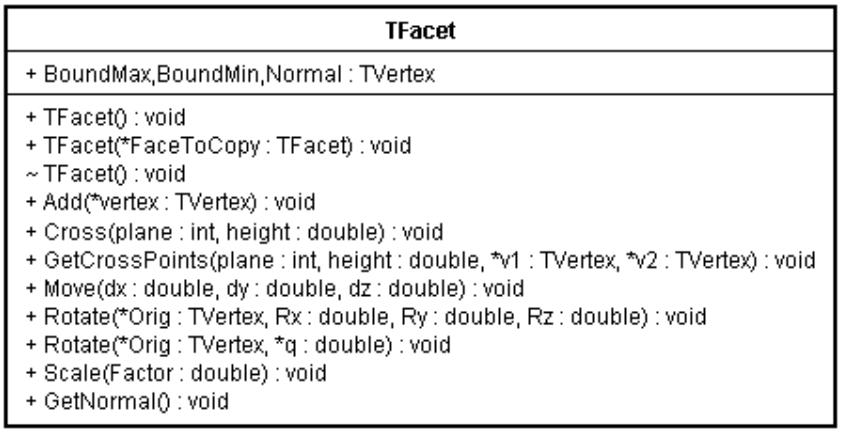

Figura A.9. Classe TFacet

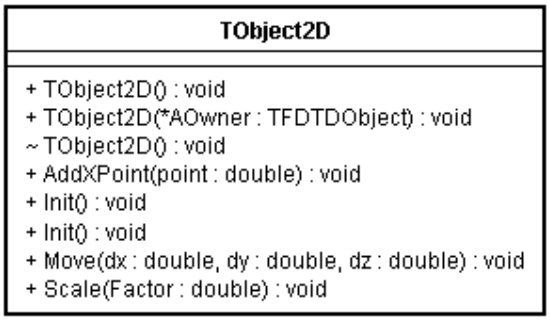

Figura A.10. Classe Tobject2D

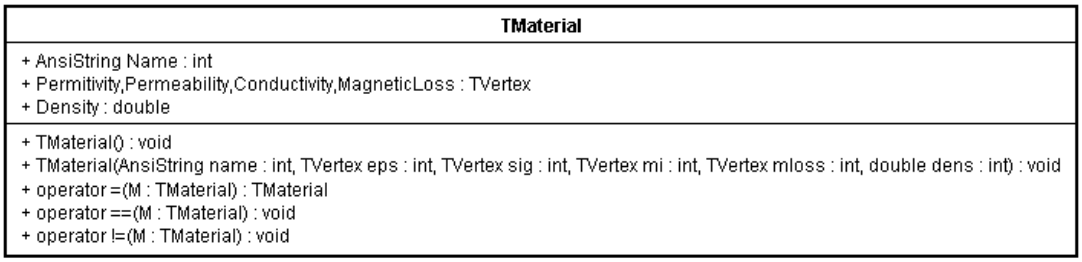

Figura A.11. Classe TMaterial

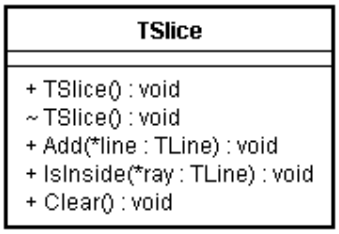

Figura A.12. Classe TSlice

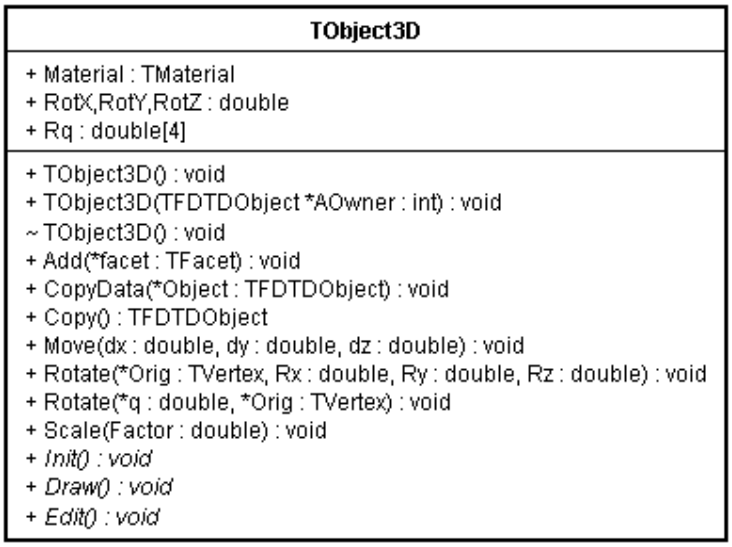

Figura A.13. Classe Tobject3D

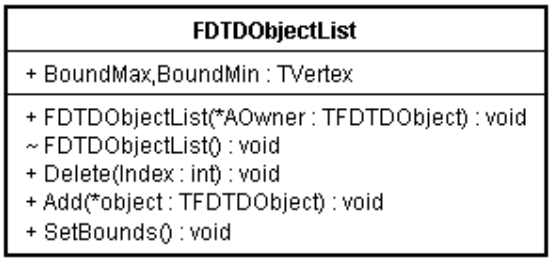

Figura A.14. Classe FDTDObjectList

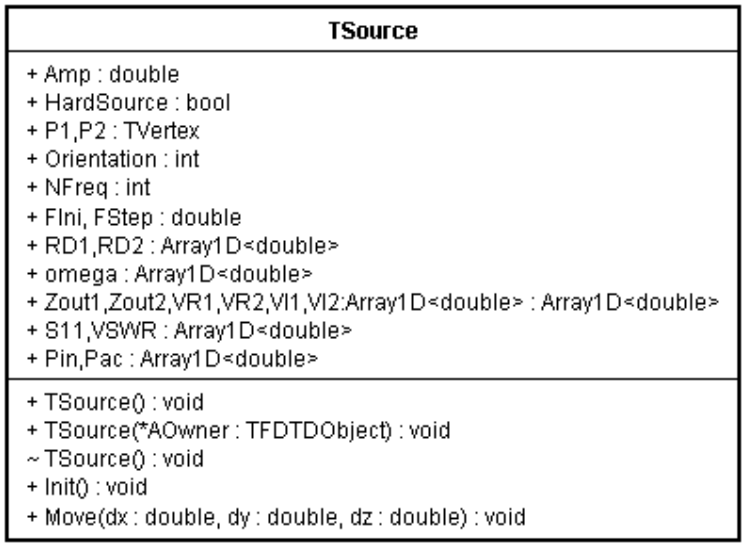

Figura A.15. Classe TSource

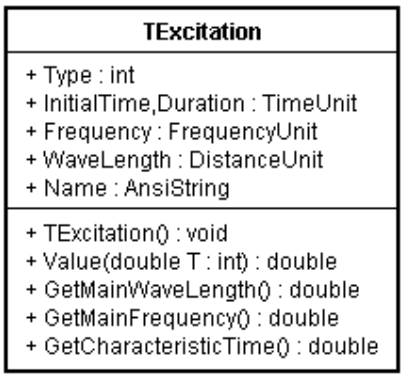

Figura A.16. Classe Texcitation

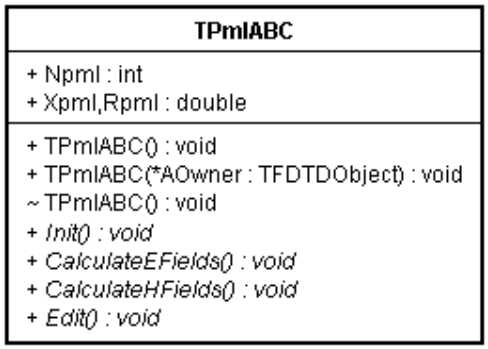

Figura A.17. Classe TPmlABC

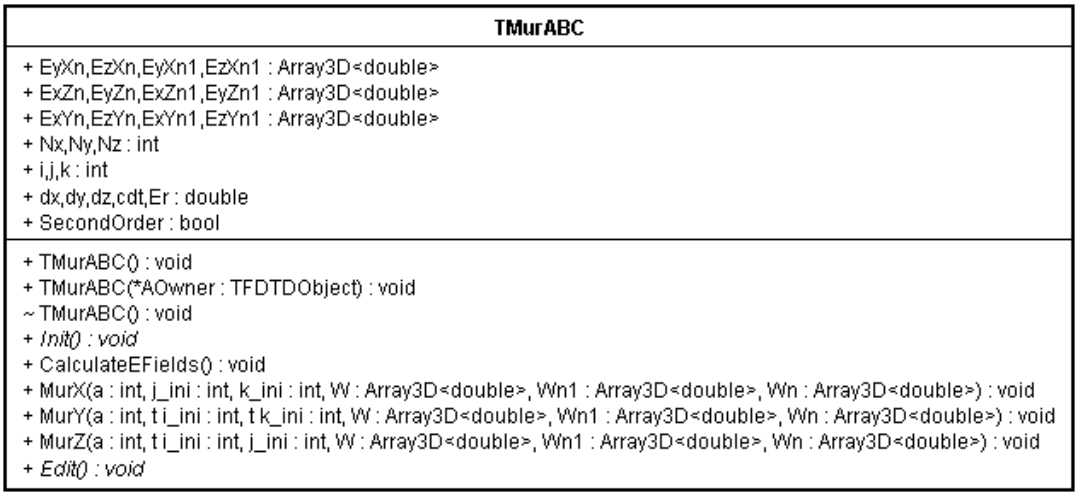

Figura A.18. Classe TMurABC

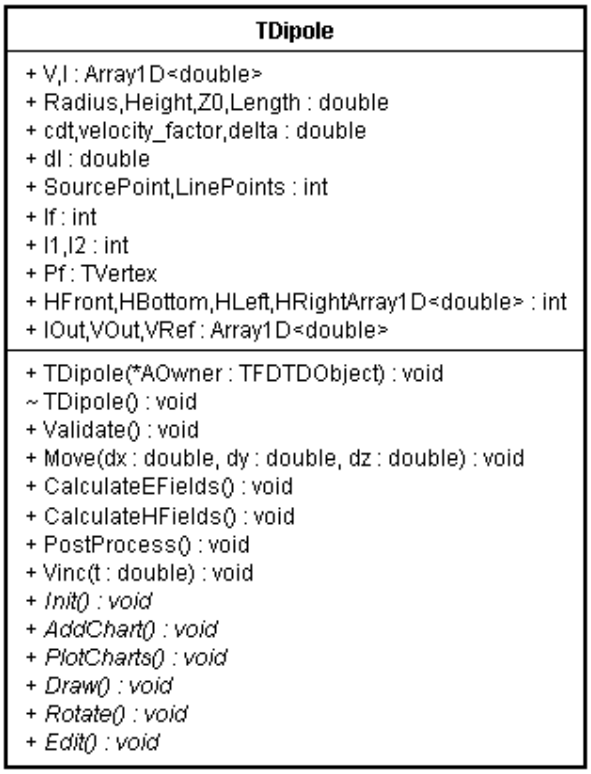

Figura A.19. Classe Tdipole

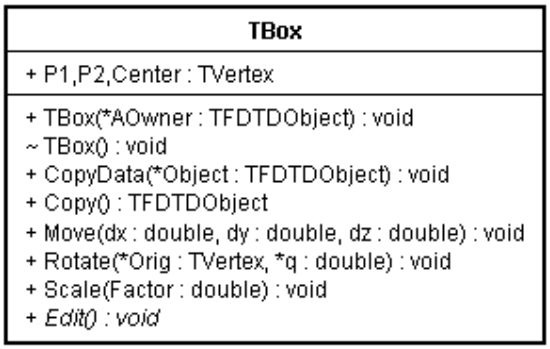

Figura A.20. Classe TBox

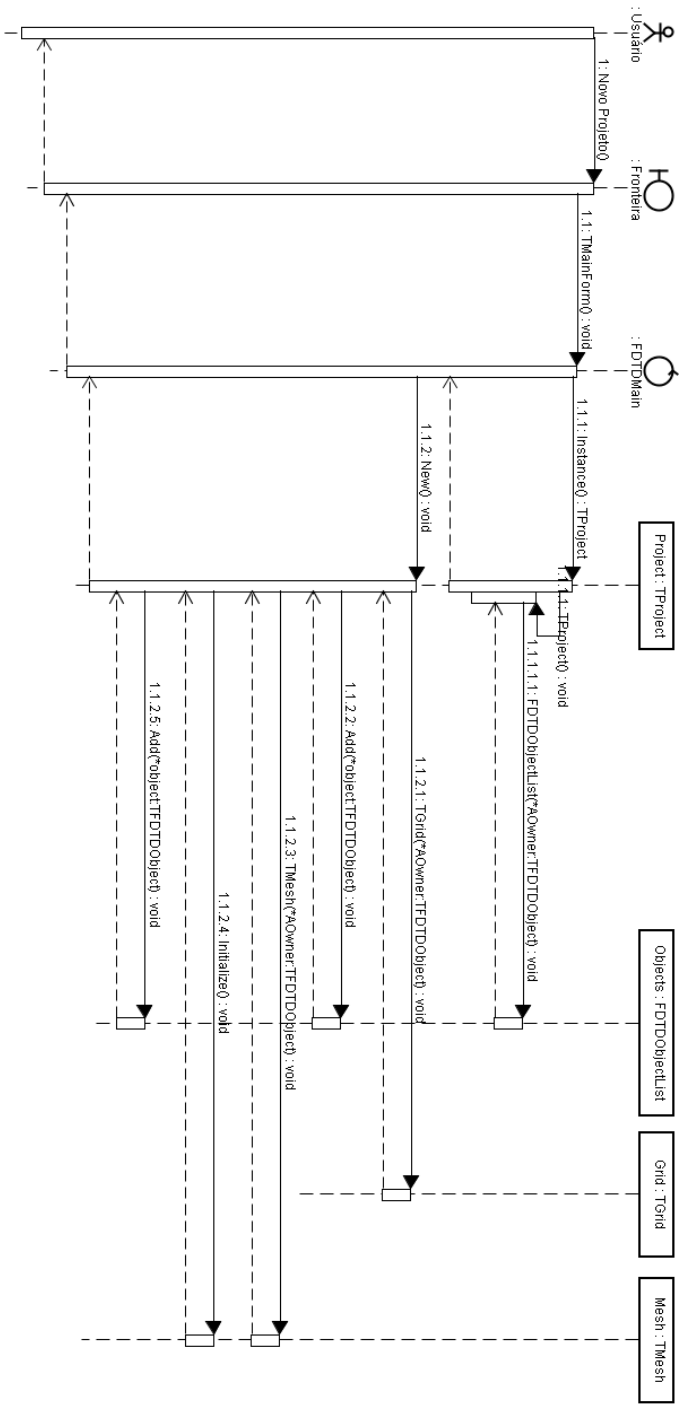

Figura A.21. Diagrama de sequência da criação de um novo projeto

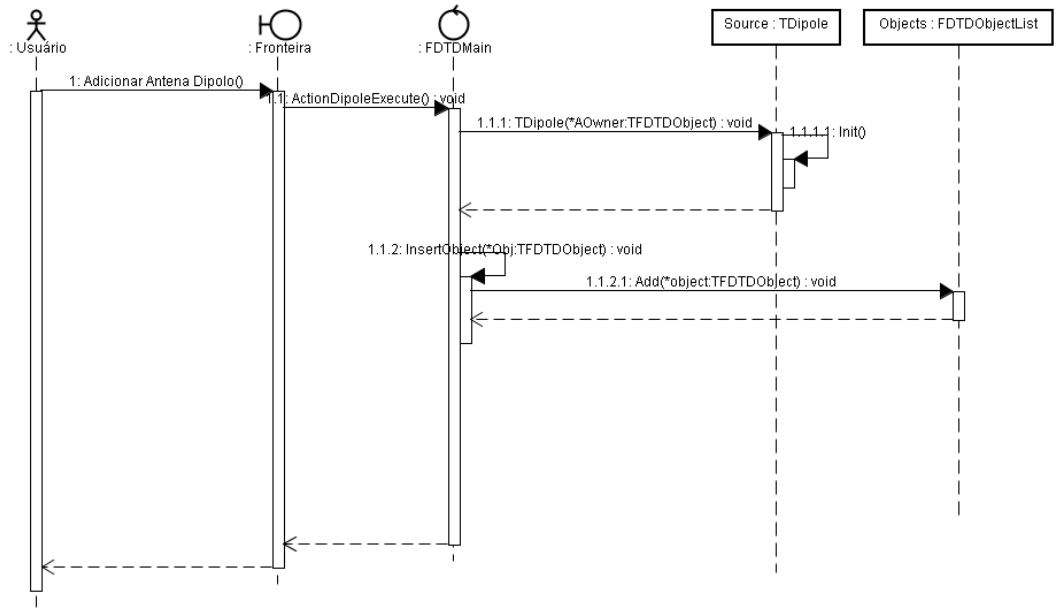

Figura A.22. Diagrama de sequência da adição da fonte Dipolo

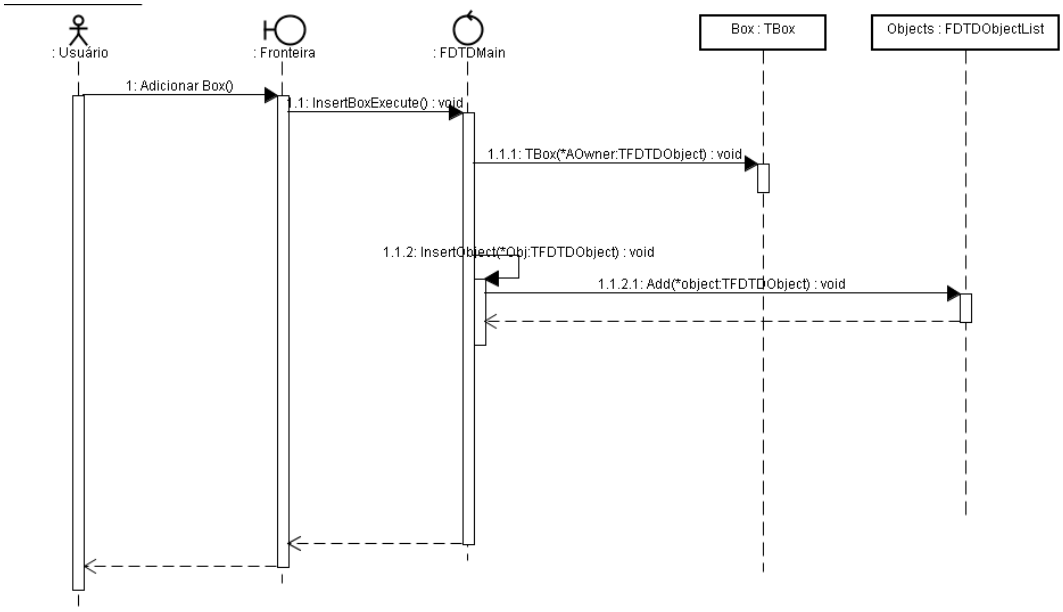

Figura A.23. Diagrama de sequência da adição do objeto Box

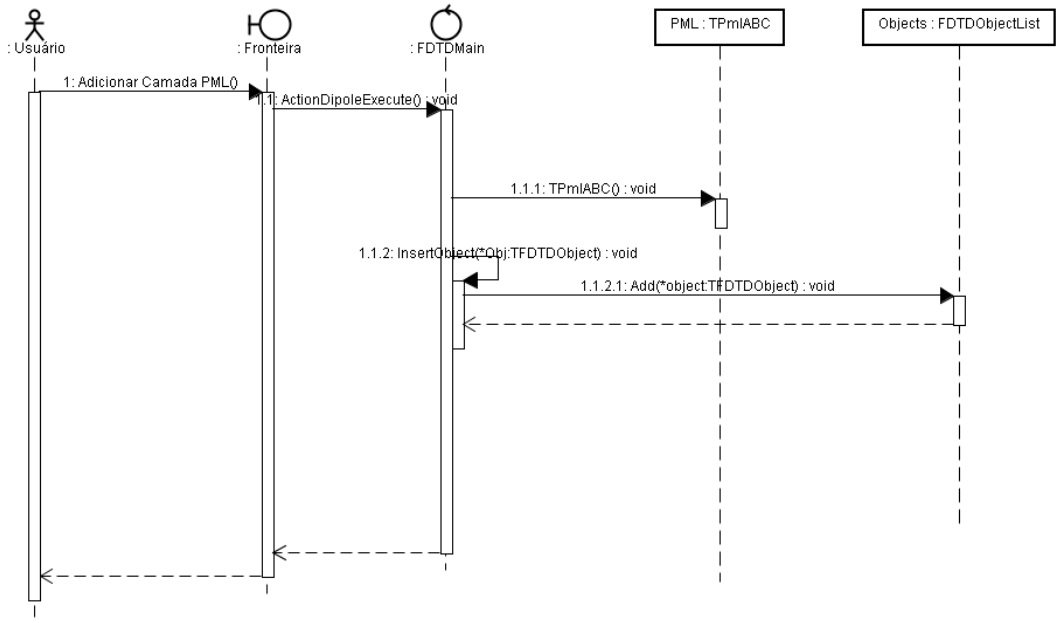

Figura A.24. Diagrama de sequência da adição da fronteira do tipo Pml

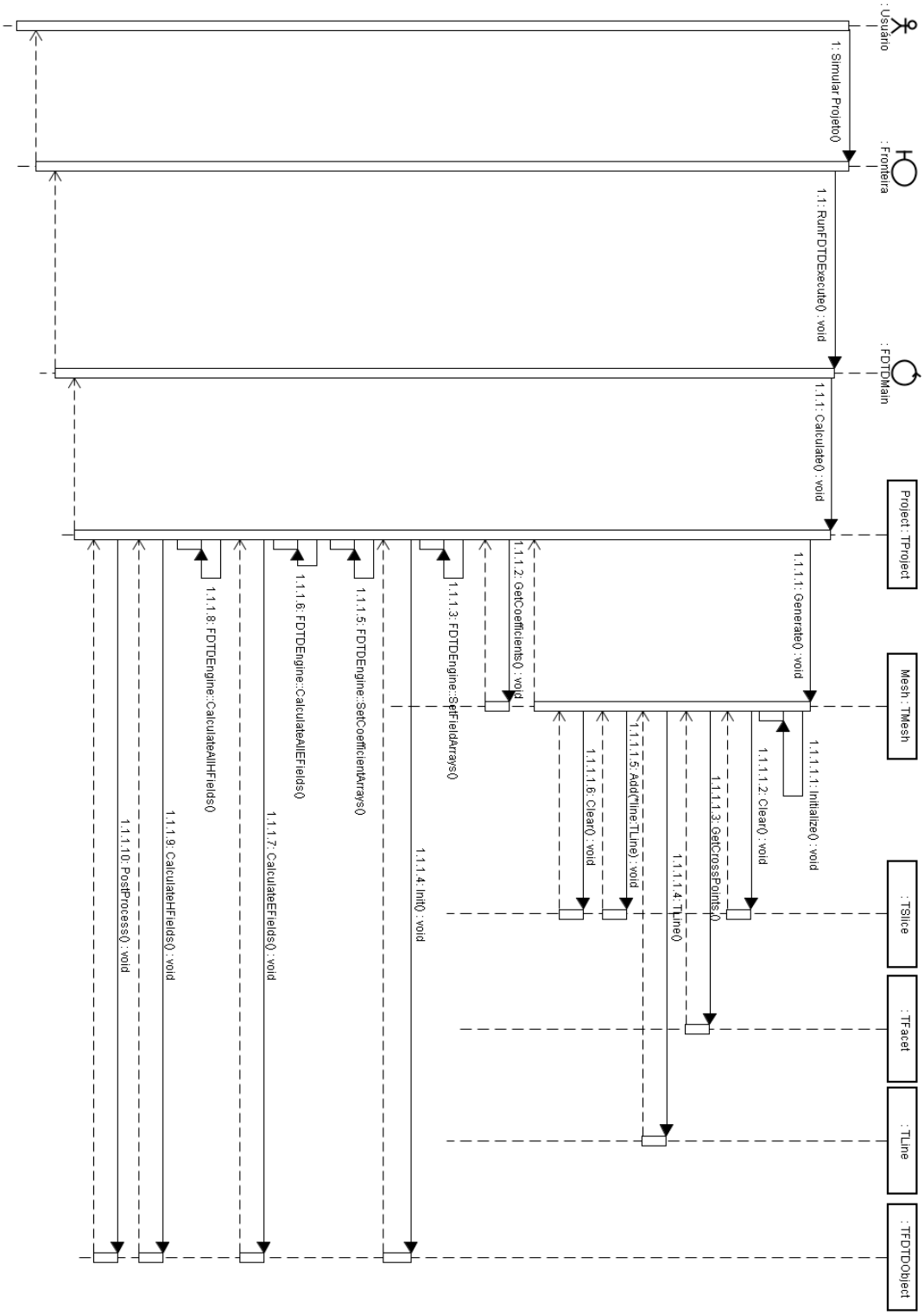

Figura A.25. Diagrama de sequência da simulação do projeto

APÊNDICE B

A documentação dos casos de uso mostrados no capítulo 4 encontram-se nas tabelas de B1 à B9.

Tabela B.1. Documentação do Caso de Uso Abrir Projeto

Nome do Caso de Uso	**Abrir Projeto**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário abra um projeto.
Pré-condições	Existir um projeto válido.
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção Abrir Projeto	
	2. Exibir ao usuário o diretório defaut.
3. Percorrer os diretórios, selecionar o projeto desejado e clicar em abrir.	
	4. Abrir o arquivo selecionado.
Restrições/Validações	O projeto deve estar em um formato adequado.

Tabela B.2. Documentação do Caso de Uso Salvar Projeto

Nome do Caso de Uso	**Salvar Projeto**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário salve um projeto.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção Salvar.	
	2. Salvar o projeto desejado.
Restrições/Validações	

Tabela B.3 . Documentação do Caso de Uso Adicionar Objeto

Nome do Caso de Uso	**Adicionar Objeto**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário adicione um objeto.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de inserir objetos.	
	2. Mostrar ao usuário os abjetos disponíveis para inserir.
3. Selecionar o objeto desejado.	
	4. Exibir as características defaut do objeto selecionado.
5. Modificar as características do objeto como desejado e confirmar.	
	6. Se as características modificadas forem incoerentes, exibir mensagem de erro mostrando as incoerências existentes e não adicionar objeto.
	7. Se as características modificadas forem coerentes, adicionar no domínio computacional o objeto requerido com as características desejadas.
Restrições/Validações	O objeto precisa possuir propriedades válidas para ser adicionado.

Tabela B.4. Documentação do Caso de Uso Adicionar Fonte

Nome do Caso de Uso	**Adicionar Fonte**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário adicione uma fonte.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de inserir fonte.	

	2. Mostrar ao usuário as fontes disponíveis para inserir.
3. Selecionar a fonte desejada.	
	4. Exibir as características defaut da fonte selecionada.
5. Modificar as características da fonte como desejado e confirmar.	
	6. Se as características modificadas forem incoerentes, exibir mensagem de erro mostrando as incoerências existentes e não adicionar fonte.
	7. Se as características modificadas forem coerentes, adicionar no domínio computacional o objeto requerido com as características desejadas.
Restrições/Validações	As características da fonte devem ser coerentes para que seja adicionada.

Tabela B.5. Documentação do Caso de Uso Adicionar Fronteira

Nome do Caso de Uso	**Adicionar Fronteira**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário adicione uma fronteira.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de inserir fronteira.	
	2. Mostrar ao usuário as fronteiras disponíveis para inserir.
3. Selecionar a fronteira desejada.	
	4. Exibir as características defaut da fronteira selecionada.
5. Modificar as características da fronteira como desejado e confirmar.	
	6. Se as características modificadas forem incoerentes, exibir mensagem de erro mostrando as incoerências existentes e não adicionar fronteira.
	7. Se as características modificadas forem coerentes, adicionar a fronteira com as características desejadas.
Restrições/Validações	

Tabela B.6. Documentação do Caso de Uso Determinar Parâmetros de Simulação

Nome do Caso de Uso	**Determinar Parâmetros de Simulação**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário determine os parâmetros de simulação de um projeto.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção grid e ajustar parâmetros relacionados aos limites de X,Y e Z e ao número de células da malha tridimensional.	
	2. Se os parâmetros ajustados forem incoerentes, exibir mensagem de erro mostrando as incoerências e não alterar os parâmetros defaut do programa.
	3. Se os parâmetros ajustados forem coerentes, salvar as modificações realizadas e ajustar a malha tridimensional adequadamente.
4. Selecionar a opção de ajuste do tempo de simulação e realizar as modificações necessárias.	
	5. Salvar as modificações realizadas.
Restrições/Validações	

Tabela B.7. Documentação do Caso de Uso Simular Projeto

Nome do Caso de Uso	**Simular Projeto**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário simule um projeto.
Pré-condições	Existir ao menos uma fonte no domínio computacional.
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de simulação.	

	2. Se não existir alguma fonte presente no domínio computacional, parar processo e exibir mensagem de erro solicitando que o usuário insira uma fonte.
	2. Determinar, pelo método do cruzamento de raios, a malha tridimensional em que está presente os objetos.
	3. Ajustar adequadamente os parâmetros de todas as células do domínio computacional.
	4. Calcular tensão de saída, corrente de saída, impedância, VSWR, perdas de retorno e potencia de entrada, levando em consideração o tipo de fonte e as condições de fronteira utilizadas. Armazenar os resultados.
	5. Se necessário, calcular o diagrama de radiação.
Restrições/Validações	É preciso existir ao menos uma fonte para que os itens 4 e 5 sejam realizados efetivamente.

Tabela B.8. Documentação do Caso de Uso Exibir Resultados

Nome do Caso de Uso	**Exibir Resultados**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas necessárias para que um usuário exiba os resultados.
Pré-condições	Ao menos uma simulação deve ter sido realizada.
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de exibir resultados.	
	2. Mostrar ao usuário os resultados existentes para exibição.
3. Selecionar o resultado desejado.	
	4. Exibir o resultado selecionado.
Restrições/Validações	Este caso de uso apenas poderá ser executado se ao menos uma simulação tiver sido feita.

Tabela B.9. Documentação do Caso de Uso Interagir com o Modelo

Nome do Caso de Uso	Interagir com o Modelo
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as etapas

	necessárias para que um usuário interaja com o modelo presente no domínio comutacional.
Pré-condições	
Pós-condições	
Ações do ator	**Ações do Sistema**
1. Selecionar a opção de interação desejada.	
	2. Fornecer ao usuário a capacidade de interação requerida.
3. Interagir com o modelo.	
	4. Modificar o modelo conforme as interações feitas.
Restrições/Validações	