UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ESTRUTURAS

UM FRAMEWORK PARA AUTOMAÇÃO/INTEGRAÇÃO DO PROCESSO DE DESENVOLVIMENTO DE PROJETOS DE ESTRUTURAS RETICULADAS TRIDIMENSIONAIS

REGINA CÉLIA GUEDES LEITE

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ESTRUTURAS

"UM FRAMEWORK PARA AUTOMAÇÃO/INTEGRAÇÃO DO PROCESSO DE DESENVOLVIMENTO DE PROJETOS DE ESTRUTURAS RETICULADAS TRIDIMENSIONAIS"

Regina Célia Guedes Leite

Tese apresentada ao Programa de Pós-Graduação em Engenharia de Estruturas da Escola de Engenharia da Universidade Federal de Minas Gerais, como parte dos requisitos necessários à obtenção do título de "Doutor em Engenharia de Estruturas".

Comissão Examinadora:

_____
Prof. Dr. José Ricardo Queiroz Franco
DEES - UFMG - (Orientador)

_____
Prof. Dr. Felicio Bruzzi Barros
DEES – UFMG

_____
Prof. Dr. Renato Cardoso Mesquita
DEE – UFMG

_____
Prof. Dr. Philippe Remy Bernard Devloo
UNICAMP

_____
Prof. Dr. Luiz Fernando Campos Ramos Martha
PUC/Rio

Belo Horizonte, 06 de agosto de 2007

Dedico este trabalho

À herança que Deus me confiou,

A aquele que tem me despertado para o verdadeiro valor da vida,

A aquele que tem me ensinado a ser mãe, a ser filha.

Meu pequeno, GRANDE, Lucas.

AGRADECIMENTOS

Agradeço a Aquele que do barro me formou, e que constantemente me ensina que, a perfeição e a justiça estão nas mãos do Oleiro, que molda o barro a cada dia, com paciência, dedicação e muito amor. Meu amado e querido Deus.

Ao Professor José Ricardo Queiroz Franco, pela orientação, pela exigência, pela compreensão e pelo apoio.

Ao meu amado esposo, Edmilson, pela direção inicial, sem a qual hoje este trabalho não existiria, e por tantas horas de apoio, de incentivo, de orientações e de consolo.

Ao meu pai e à minha mãe pelo grande apoio, e pelas horas e horas de imensurável ajuda.

Ao meu irmão pelo seu exemplo e por instantes de breve, mas valiosa orientação.

Ao professor Armando C. Campos Lavall, ao professor Marcelo Greco e à Renata L. Silva pelos programas cedidos para aplicação nesta pesquisa e pelo suporte na utilização dos mesmos.

Ao professor Flávio Antônio dos Santos, Diretor-geral do CEFET-MG, que acreditou e investiu nesta pesquisa, criando condições para que a mesma fossa concluída.

A todos do grupo CADTEC por suas contribuições para este trabalho.

À Fundação de Amparo à Pesquisa do Estado de Minas Gerais, FAPEMIG, pela bolsa de doutorado a mim concedida.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ADN - Autodesk Developer Network

API - Aplication Program Interface

ARX - AutoCAD Runtime Extention

BRFEM - Brazil Finite Elements

DBX - Database Extension

CAD - Computer Aided Design

CADTEC - Centro de Apoio, Desenvolvimento Tecnológico e Ensino da Computação Gráfica

CAE - Computer Aided Engineering

CAM -Computer Aided Manufacturing

CIM - Computer Integrated Manufacturing

CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico

GOPAC – Grupo de Otimização de Projeto Assistido por Computador

GUI - Grafical Users Interface

INSANE - Interactive Structural Analysis Environment

MEF – Modelo de Elementos Finitos

MFC – Microsoft Foundation Classes

MVC - Model-View-Controller

POO – Programação Orientada a Objetos

REMFRame – Reticular Structural Modeler Framework

RESUMO

Esta tese apresenta a arquitetura, o projeto e a implementação de um *framework* destinado à geração de sistemas de integração aplicados a processos de projetos de estruturas reticuladas. Estes sistemas permitem a automação do processo de desenvolvimento de projetos de estruturas através da integração entre as tecnologias CAD (*Computer Aided Design*), CAE (*Computer Aided Engineering*) e CAM (*Computer Aided Manufacturing*), presentes no processo, otimizando sua eficiência, melhorando sua produtividade e a qualidade do produto final.

Este *framework* foi projetado segundo os padrões de arquitetura MVC (*Model-View-Controller*) e *Microkernel*, com a utilização de diversos padrões de projeto reconhecidos na literatura. A arquitetura do *framework* foi projetada para abstrair as informações essenciais contidas em estruturas reticuladas. Estes dados são armazenados e gerenciados para alimentar os sistemas externos envolvidos no processo. O *framework* é definido, ainda, pelo relacionamento entre objetos e classes, implementando o funcionamento genérico dos sistemas, deixando para as aplicações a definição de comportamentos específicos. Desta forma o *framework* permite instanciar diferentes aplicativos dentro do referido domínio.

O *framework* permite esta flexibilização através de diferentes mecanismos, tais como: na interface de usuários para entrada de dados; na interface de arquivos, permitindo a importação e exportação dos dados em diferentes formatos; na criação de visualizações para os objetos modelados em diferentes plataformas gráficas; na configuração dos dados de ambiente para cada sistema integrado; na criação de objetos complexos, etc.. Desta forma, o núcleo do *framework* é capaz de manipular detalhes específicos das aplicações sem conhecê-los, baseando-se em seu funcionamento geral.

Finalmente são apresentadas algumas aplicações para diferentes áreas, obtidas através do uso do *framework,* são elas: três modeladores geométricos e cinco sistemas de integração. Estas aplicações foram implementadas com sucesso, sem a necessidade de alterações estruturais no núcleo do *framework*. Apenas alguns ajustes foram realizados no interior de classes, para melhor representar os detalhes específicos dos problemas tratados.

Palavras-chave: Framework, Padrões, Modelagem, Automação.

ABSTRACT

This thesis presents the arquitecture, design and implementation of a framework for the generating integration systems applied to the design process of reticular structures. These systems allow the automation of the structural design process by integrating the technologies CAD *(Computer Aided Design),* CAE *(Computer Aided Engineering)* and CAM *(Computer Aided Manufacturing)* used in the process, optimizing its efficiency, improving its productivity and improving the quality of the final product.

This *framework* was designed following the MVC (*Model-View-Controller*) and *Microkernel* arquitectural patterns and using several design patterns acknowledged in the literature. The *framework* arquitecture was designed to abstract the essential data contained in reticular structures. The data is stored and managed to feed external systems involved in the process. The *framework* is also defined by the relationship among the objects and classes, which define the general functionalities for the systems, and allowing the applications to define their specific behavior. The *framework*, thus, allows the development and the instanciation of different applications inside its domain.

The *framework* provides flexibility through different mechanisms, such as: through the user interface of input data; through the files interface, allowing data to be imported and exported in different formats; through the visualisation of objects modelled by different graphic platforms; through the configuration of the environment data for each integrated system; through the creation of complex objects, etc.. Therefore, the *framework's* core is capable of manipulating specific data of applications, without any knowledge of their details, but based on the general behavior defined.

Finally, this thesis presents some applications implemented using this *framework* and applied in different areas, as: three geometric modelers and five integration systems. These applications were successfully implemented without any structural changes in the framework core. Only some adjustments inside some classes were made, to better represent specific details of the problems studied.

Keywords: *Framework, Patterns, Modelling, Automation.*

1

INTRODUÇÃO

Este capítulo apresenta uma descrição sumária da presente tese, contextualizando-a com relação a sua área de aplicação. Em seguida são definidos seu objetivo geral, a hipótese adotada e seus objetivos específicos. Finalmente é apresentada a organização da tese em capítulos.

Esta pesquisa está vinculada ao Curso de Doutorado do Programa de Pós-Graduação em Engenharia de Estruturas da Escola de Engenharia da UFMG, na área de concentração Métodos Numéricos e Computacionais para Engenharia, sub-área relacionada com as tecnologias CAD/CAE/CAM/CIM[1]. O Projeto Assistido por Computador (CAD), encadeado com a Engenharia e a Manufatura Assistida por Computador (CAE/CAM), são tecnologias que podem ser gerenciadas através da Manufatura Integrada por Computador (CIM) para automatizar processos industriais.

Esta tese tem como objetivo contribuir, de forma inovadora, com estudos científicos desenvolvidos na avaliação e na implementação de ferramentas computacionais aplicadas ao processo produtivo de projetos de Engenharia de Estruturas. Dentro deste objetivo são apresentados os resultados obtidos pelo desenvolvimento de uma ferramenta computacional cujo enfoque é a integração entre etapas desse processo. Através desta integração pretende-se contribuir para um aumento da qualidade[2] do processo pela automação da comunicação entre procedimentos e dados oriundos de sistemas independentes aplicados em cada uma das etapas integradas.

Mais especificamente, esta pesquisa foca o desenvolvimento de projetos de estruturas reticuladas. Entende-se por projeto de estruturas reticuladas um conjunto de procedimentos necessários para viabilizar a execução de uma estrutura caracterizada pela presença de elementos lineares. São elementos lineares ou prismáticos os

---

[1] *Computer Aided Design* (CAD), *Computer Aided Engineering* (CAE), *Computer Aided Manufacturing* (CAM) e *Computer Integrated Manufacturing* (CIM)

[2] Geração de produtos e serviços adequados aos clientes do processo, satisfazendo as necessidades dos clientes.

elementos estruturais onde uma de suas dimensões, normalmente o comprimento, é relativamente maior do que as demais dimensões: largura e altura da seção transversal.

Dentre os procedimentos que compõem um projeto de estruturas reticuladas esta pesquisa teve como foco os seguintes: o lançamento geométrico dos elementos estruturais, a análise de esforços, o dimensionamento das seções transversais, o dimensionamento e o detalhamento de ligações, a confecção das pranchas de desenho para fabricação e montagem ou execução. Uma outra etapa de fundamental importância, que viabiliza a execução do projeto e pode ser inserida no conjunto de procedimentos do processo de um projeto estrutural é o orçamento. O orçamento da estrutura de uma edificação é uma etapa passível de integração ao processo, uma vez que os dados necessários para sua execução podem ser facilmente extraídos do conjunto de dados gerados e gerenciados durante a etapa de modelagem/concepção da estrutura.

Esta pesquisa tem como objetivo geral o desenvolvimento de um sistema gerenciador de dados capaz de promover a integração de etapas do processo de projeto de estruturas reticuladas, visando contribuir para a melhoria da produtividade, da qualidade do processo e do produto.

Este objetivo geral foi delineado tendo como hipótese a seguinte assertiva:

Um projeto de estruturas reticuladas é desenvolvido através de um processo composto por diversas etapas. Este processo constitui um domínio de aplicação, no qual é possível identificar um núcleo de dados e de procedimentos comuns a diferentes projetos. Este núcleo de dados é compartilhado pelas etapas de desenvolvimento. Este

horizonte aponta para a possibilidade do desenvolvimento de um *framework*[3] a partir do qual é possível gerar diferentes sistemas de integração das etapas desse processo.

Como objetivos específicos desta pesquisa, podemos listar:

1. Desenvolver o *framework* a partir da identificação clara de um domínio de aplicações dentro do processo de projeto de estruturas reticuladas. Apresentar uma abstração deste domínio com o uso dos conceitos de Engenharia de *Software* sob o paradigma da Orientação de Objetos, uso de padrões de projeto e de arquitetura, buscando-se obter um sistema robusto, confiável e re-aproveitável.

2. Garantir a possibilidade de aplicar o *framework* no desenvolvimento de sistemas que possam utilizar diferentes plataformas gráficas para visualização dos dados tratados. Para responder a este objetivo torna-se necessário garantir uma independência entre dados, controle de fluxo e visualização de resultados.

3. Validar o *framework* através do instanciamento de aplicações.

4. Avaliar o *framework* sob o ponto de vista de sua potencialidade para o desenvolvimento de novas pesquisas relacionadas ao mesmo.

Assim, o *framework* tem como missão primeira proporcionar, através de uma arquitetura consistente, o desenvolvimento mais rápido e mais simples de aplicações capazes de integrar, em um único fluxo, os dados oriundos de diversas etapas do processo de projeto de estruturas e ainda interagir, através de interfaces de arquivos de

---

[3]*Framework* é definido em linhas gerais como um sistema generalista utilizado para o desenvolvimento de outros sistemas especialistas dentro de um domínio de aplicações.

entrada e de saída de dados, com sistemas independentes necessários para cumprir o ciclo do processo.

Esta Tese está organizada em seis capítulos. No Capítulo 2 é apresentada uma reflexão sobre o projeto de estruturas, objetivando elencar os motivos que levaram à busca pela unificação do processo identificado. Nesse capítulo são identificadas e listadas as etapas de projeto, suas características, os sistemas aplicáveis a cada etapa e a relação entre estas etapas.

A seguir, no Capítulo 3, apresenta-se a *Revisão Bibliográfica* realizada sobre sistemas utilizados na modelagem de estruturas reticuladas e sistemas que aplicam a Programação Orientada a Objetos (POO) na implementação computacional do Método dos Elementos Finitos (MEF). O estudo sobre as arquiteturas e soluções propostas serviu como embasamento para a definição e estruturação do que passaremos a chamar de REMFrame (*Reticular Structural Modeler Framework*), um *framework* para a modelagem de estruturas reticuladas. Esta etapa da pesquisa permitiu a identificação da necessidade de reunir em uma solução generalista os conhecimentos e os esforços apresentados em diferentes pesquisas aplicadas ao mesmo domínio. Esta síntese de soluções e decisões objetiva evitar que estas se percam com o tempo, tornando tais soluções disponíveis através de uma base confiável, consistente e re-aproveitável.

O Capítulo 4 trata da *Metodologia de Pesquisa* apresentando o Referencial Teórico e o Delineamento Metodológico. No Referencial Teórico são apresentados os conceitos referentes ao objeto de pesquisa e às teorias utilizadas integralmente ou parcialmente no desenvolvimento do mesmo. No Delineamento Metodológico são abordados os conceitos específicos, os procedimentos, os instrumentos e as ferramentas utilizadas e as técnicas de pesquisa adotadas.

O Capítulo 5, por sua vez, apresenta a arquitetura do REMFrame, os principais casos de uso e os *hot spots* definidos.

O Capítulo 6 apresenta algumas aplicações desenvolvidas a partir do REMFrame, testes de aplicação destes sistemas e estudos de caso de lançamento estrutural.

Finalmente, o Capítulo 7 apresenta as principais conclusões inferidas sobre os resultados obtidos e sobre o alcance dos mesmos e as sugestões para novos trabalhos. A este capítulo, seguem o Referencial Bibliográfico, a Bibliografia Consultada e o Glossário.

2

O PROCESSO DE DESENVOLVIMENTO DE PROJETO DE ESTRUTURAS RETICULADAS

Este capítulo aborda inicialmente a importância da vinculação entre as etapas do processo de projeto de estruturas reticuladas para a qualidade final do produto. Em seguida é definido, em linhas gerais, o projeto de estruturas reticuladas. Finalmente são apresentados os conceitos de modelagem estrutural e modelagem geométrica.

2.1 A Vinculação Entre as Etapas do Processo de Projeto e a Qualidade do Produto Final

Gerir a qualidade nas fases projeto reduz a probabilidade de existirem problemas latentes que aparecem durante toda a vida do produto. A prevenção é melhor, e mais barata do que a cura (SOUZA, 1995).

O processo de projeto é entendido não só como a concepção de um produto, mas como o processo que determina todas as especificações de formas, dimensões, materiais, componentes e elementos construtivos relativos às exigências do usuário. O processo de desenvolvimento de projetos é composto por um grande número de outros processos sob responsabilidade de diversos agentes. Cada agente é responsável pela qualidade do produto-projeto que lhe diz respeito evitando defeitos de projeto ou não conformidade com as necessidades do cliente, quer seja este um agente interno ao processo ou um agente externo (outros processos ou o usuário final).

Segundo VARALLA (2003) o projeto é a etapa mais importante dentro do processo do planejamento, pelas implicações que ela tem na definição do produto e do processo da produção. A falta de integração entre as atividades de um processo tem como conseqüências interrupções não desejadas, aumentos de custos e o comprometimento da qualidade do produto. O estabelecimento das coordenações necessárias no processo de definição do produto reduz os custos finais por falhas no projeto. Investir tempo e recursos na fase de projeto permite gerar melhores soluções.

As soluções adotadas na etapa de projeto têm amplas repercussões em todo o processo da construção e na qualidade do produto final. Para assegurar a qualidade de um projeto é necessário controlar a qualidade do seu processo de elaboração. O resultado final de um processo, no qual o desenvolvimento de vários projetos é realizado de forma separada, não representa a solução mais adequada em termos de grau de complexidade, continuidade e de desempenho. Se um projeto não realimenta os demais ao longo do processo as dificuldades podem se propagar (SOUZA, 1995 e SILVA E SOUZA, 2003).

A construção civil tem apresentado, nas últimas décadas, um constante aprimoramento de técnicas, de processos, de procedimentos e de metodologias aplicadas. São notórios os avanços tecnológicos nas técnicas construtivas, no desenvolvimento de materiais e nos controles dos processos envolvidos. Na área de Engenharia de Estruturas testemunhamos o avanço de teorias de análise e de dimensionamento, além de avanços tecnológicos obtidos pelos métodos computacionais aplicados juntamente com estas teorias. Por outro lado, a indústria da construção civil apresenta ainda, de uma maneira geral, uma grande deficiência no aproveitamento de esforços, com a necessidade de re-trabalho desde o lançamento do empreendimento à sua execução. A falta de sincronia de informações entre etapas de um projeto, entre os diversos projetos de um empreendimento e ainda entre o projeto e a execução geram diversos erros, perdas, atrasos e custos extras ao longo da produção.

No projeto de estruturas, a falta de vinculação entre as etapas de projeto é o fator preponderante da origem de erros, atrasos e prejuízos. Apesar da existência de sistemas CAD, CAE e CAM específicos para cada uma das etapas, estes sistemas operam, em sua maioria, de forma desvinculada, sem uma integração com os demais sistemas aplicados no processo. Apesar da aplicação destas tecnologias contribuir para o aumento de precisão de cada etapa isoladamente, é exatamente na transferência de dados que a maior parte dos erros e problemas podem ocorrer.

Para a transposição de dados de uma plataforma para outra pode ser necessário uma manipulação destes dados, alterando formatações, realizando conversões de unidades, de coordenadas, etc. O fator humano inserido nestas atividades aumenta a possibilidade de erros devidos tanto à inserção incorreta de dados quanto à interpretação dos mesmos. Além disso, a re-alimentação de cada etapa com os dados alterados em

etapas seguintes aumenta a margem de erros e de problemas que podem ocorrer ao longo do processo.

A automação do ciclo de produção "planejar-projetar-fabricar" deve ir além do desenvolvimento de sofisticados sistemas de automação para cada uma destas etapas ou para suas etapas internas. Para o aumento da qualidade deste ciclo, torna-se necessário integrar estes sistemas. Além de reduzir erros, uma integração dos sistemas proporciona um aumento de produtividade[4] ao processo, diminuindo o fator tempo gasto na transferência de dados. Ao facilitar a alimentação de cada etapa com os dados das demais, a automação na transferência de dados permite que o processo possa apresentar um caráter iterativo, ou cíclico, em busca de soluções cada vez mais refinadas.

2.2 O Projeto de Estruturas Reticuladas

De uma maneira geral, o projeto de estruturas reticuladas pode ser dividido nas seguintes etapas: lançamento e modelagem da estrutura, análise estrutural, dimensionamento de elementos, dimensionamento de ligações, detalhamento de elementos, detalhamento de ligações. Estas etapas são automatizadas por sistemas CAD e CAE. As tecnologias CAD são normalmente aplicadas nas etapas de lançamento e de detalhamento das estruturas, enquanto as tecnologias CAE são aplicadas às etapas de análise e de dimensionamento. A tecnologia CAM é aplicada ao processo de fabricação na automação da produção dos elementos estruturais ou de seus componentes.

---

[4] Entende-se como produtividade “o nível de rendimento atingido na utilização de recursos. A produtividade é representada pela razão entre produto e insumo“ (SOUZA, 1995).

Atualmente a tecnologia CAM vem sendo amplamente utilizada no corte, na dobra e na soldagem de ferragem para estruturas de concreto convencional e na indústria de estruturas pré-fabricadas[5] de concreto. Na indústria de estruturas de concreto pré-moldado esta tecnologia é também utilizada na confecção da ferragem, no uso de formas deslizantes, na produção e usinagem do concreto sob um controle rigoroso de qualidade, etc. Além destas, a indústria de estruturas metálicas gradativamete automatiza seus processos de fabricação através das tecnologias CAD/CAE/CAM/CIM desde a concepção/modelagem da estrutura até o corte, furação, dobra ou perfilação de chapas utilizando máquinas digitais CNC.

A tecnologia CAD integrada aos sistemas CAE permite transformar dados numéricos em representações gráficas, que funcionam como pré e pós-processadores. Como pré-processador, ela permite a visualização do modelo correto ou a sua correção, garantindo, assim, uma interação sem erros com as fases de análise e de dimensionamento, cujos resultados poderão ser pós-processados graficamente para uma melhor, e mais fácil, análise e interpretação. A missão de um sistema de integração das etapas de um processo de desenvolvimento de projetos de estruturas é o controle e o gerenciamento de dados variados, garantindo uma importação/exportação precisa e confiável. As fases de modelagem, de análise e de dimensionamento precisam comunicar-se através de interfaces, que permitam a transferência de dados sem erros ou perdas. Da mesma forma a integração entre as fases de dimensionamento e o detalhamento precisa ser gerenciada por um sistema que garanta a integridade dos dados a serem transferidos automaticamente. Para estruturas industriais, tais como estruturas

[5] Segundo EL DEBS (2000), elementos pré-moldados são aqueles moldados ou produzidos fora do seu lugar de uso definitivo na estrutura, sendo que a pré-fabricação é a pré-moldagem aplicada à produção em grande escala.

pré-fabricadas, estruturas prediais e estruturas de torres, os sistemas devem gerenciar também a automação da integração entre interfaces das fases de detalhamento e fabricação para permitir uma alimentação precisa das máquinas digitais de manufatura automatizada.

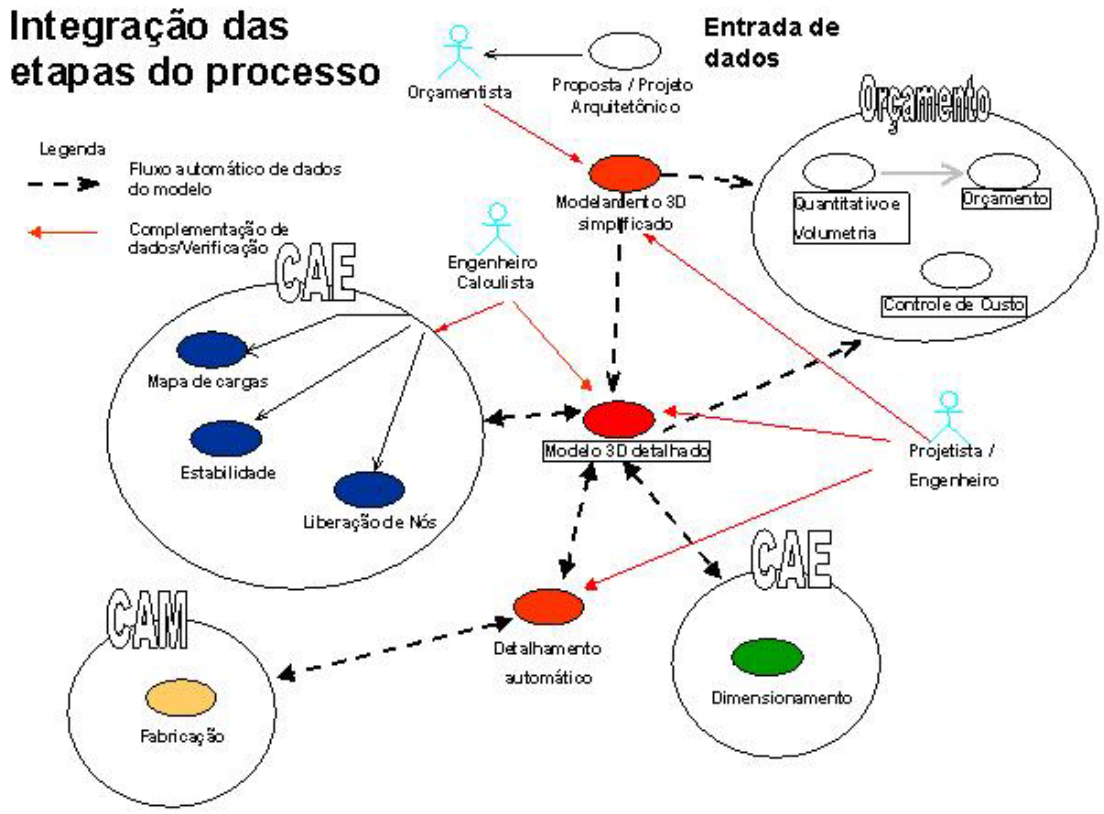

FIGURA 2.1- Fluxo geral do processo produtivo de estruturas reticuladas com o uso de um sistema de integração.

Assim, a automação de projetos de estruturas deve ser buscada não apenas através do desenvolvimento de sistemas individualizados via tecnologias CAD, CAE e CAM, mas pelo desenvolvimento de sistemas de integrados, que permitam o gerenciamento e a integridade de dados ao longo do processo para o intercâmbio de informações entre as várias etapas do mesmo (FIG. 2.1). Esses sistemas devem fornecer interfaces compatíveis entre os sistemas CAD e os sistemas CAE utilizados no projeto, podendo ainda prever uma interface para os sistemas CAM do processo de fabricação.

A conseqüente diminuição de erros e de desperdícios de tempo e de recursos humanos, obtidos pelo uso destes sistemas, vão refletir em ganhos financeiros e na competitividade da indústria. Estes sistemas de integração devem ser consistentes, completos e corretos, para garantir um alto grau de confiabilidade ao processo, a qualidade do produto e a eficiência das soluções adotadas.

2.3 Modelagem Estrutural e Modelagem Geométrica

Um modelo é definido como uma abstração de um objeto ou de um processo real. "Um modelo é um objeto construído artificialmente com a finalidade de tornar mais fácil a observação de um outro objeto" (MÄNTYLÄ[6] *apud* MAGALHÃES, 2000) No modelo o elemento real é representado por um conjunto de dados fundamentais, que o representam e o distinguem da sua vizinhança, ou do seu ambiente. O modelo de objeto, definido pela POO é a representação computacional de um objeto, sendo formado pelo conjunto de dados necessários para representar um elemento real em um sistema orientado a objetos.

Um modelo geométrico armazena os dados que definem a geometria de um objeto. A geometria de um objeto é fundamental para sua reprodução através de um sistema gráfico ou através de um processo produtivo. A modelagem geométrica estuda a representação computacional da geometria de um objeto, tratando também das demais

---

[6] MÄNTYLÄ, M. An introduction to solid modelind. Maryland: Computer Science, 1988.401p.

informações necessárias para suportar o projeto, a análise e a fabricação de produtos. Segundo MÄNTYLÄ[5] a modelagem geométrica pode ser dividida em:

- Modelagem gráfica: representa os objetos através de arestas e vértices, sem fazer referência a faces ou a superfícies. Gera os modelos fio-de-arame ou *wireframe*;

- Modelagem de superfície: suporta informações sobre superfícies planas ou curvas, mas não reconhece sólidos;

- Modelagem de sólidos: descreve a geometria tridimensional de um objeto como um sólido.

Os modelos sólidos podem armazenar a caracterização completa de um objeto. A partir de simplificações do modelo sólido é possível obter os modelos de superfície e de fio-de-arame.

Modelagem estrutural é a representação de uma estrutura através de um modelo estrutural. Este modelo deve ser capaz de representar a geometria da estrutura real e suas condições de contorno. Além disso, ele deve ser capaz de simular o comportamento físico de uma estrutura submetida a esforços, respondendo aos mesmos em relação à resistência dos elementos e em relação aos deslocamentos apresentados.

A modelagem estrutural deve ser capaz de representar de forma simples e organizada diversos tipos de estruturas, armazenando os dados necessários à sua análise e ao seu dimensionamento.

No modelo estrutural, as características geométricas da estrutura e outros dados, essenciais à análise e ao dimensionamento são representados por dados numéricos. A alimentação de muitos sistemas de análise e de dimensionamento é feita diretamente através dos dados numéricos inseridos em algum tipo de formulário pelo usuário. Esta alimentação manual de dados gera uma considerável probabilidade de erros como mencionado anteriormente.

Com o desenvolvimento da computação gráfica, alguns programas de análise incorporaram ferramentas gráficas para a definição da geometria do objeto estudado, são os pré-processadores. Alguns destes programas também possuem recursos gráficos

para apresentação e interpretação de resultados da análise, os pós-processadores. Uma entrada de dados via processador gráfico possibilita a visualização do modelo estrutural, permitindo uma conferência visual dos dados inseridos. A automação de critérios de controle, como restrições ou validações sobre os dados inseridos, reduz a probabilidade de erros. O pós-processamento permite ao usuário uma avaliação visual do comportamento da estrutura, o que facilita a identificação de pontos críticos ou de falhas de projeto. Estas informações são, então, utilizadas para o aprimoramento da concepção estrutural. Estes sistemas são, usualmente, aplicativos comerciais aplicados em uma etapa do projeto de estruturas, a análise ou o dimensionamento, ou aplicações específicas para algum um tipo de estrutura: estruturas em concreto armado, estruturas metálicas pré-fabricadas prediais, estruturas de galpões em concreto pré-moldado, torres de transmissão, etc.

Um modelo unifilar é um modelo *wireframe* utilizado na etapa de análise estrutural de uma edificação. Este modelo faz a representação de uma estrutura através de um conjunto de nós e barras representadas pelos seus eixos. As barras representam os elementos lineares ou prismáticos, geralmente vigas e pilares. Os nós representam as ligações entre elementos lineares. Neste modelo os elementos são submetidos a condições de contorno. As barras podem receber carregamentos, liberações e deformações iniciais. O nós recebem, além dos carregamentos, restrições ao deslocamento ou à rotação e deslocamentos prescritos. Além disso, para a análise das estruturas, outros dados devem ser inseridos no modelo estrutural: as propriedades físicas e geométricas dos elementos. Estes dados variam com a teoria de análise adotada e com o tipo de elemento utilizado.

Para o detalhamento gráfico de elementos e de ligações, o modelo estrutural pode ser representado por um modelo de superfície ou por um modelo sólido. Estes modelos são capazes de armazenar os dados do modelo unifilar, além de dados sobre a geometria das seções transversais e ainda dados sobre a geometria das ligações entre elementos.

3

REVISÃO BIBLIOGRÁFICA

Este capítulo segue a seguinte estruturação:

Na introdução é apresentado o grupo de pesquisas CADTEC (Centro de Apoio, Desenvolvimento Tecnológico e Ensino da Computação Gráfica) sendo descrito, em linhas gerais, o trabalho realizado pelo mesmo. Em um segundo momento são apresentados os dois domínios de aplicação abordados pela revisão bibliográfica e os trabalhos estudados em cada um dos domínios. Os itens seguintes apresentam uma síntese do estudo detalhado dos trabalhos considerados relevantes para esta tese, seguido imediatamente por uma análise comparativa dos pontos de cada trabalho, aplicáveis à presente pesquisa.

3.1 Introdução

O CADTEC é um grupo de pesquisa certificado pelo CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), vinculado ao Departamento de Engenharia de Estruturas da UFMG. O grupo foi fundado em 1997, e conta atualmente com nove pesquisadores, seis estudantes e dois técnicos.

Desde 1997 o Grupo CADTEC tem trabalhado na automação de processos via tecnologia CAD/CAE/CAM/CIM. Neste período foram desenvolvidos onze sistemas independentes para a automação das etapas de modelagem e/ou detalhamento de estruturas, sendo que dois sistemas incluíram também a etapa de análise estrutural. Destes, seis sistemas foram desenvolvidos através de pesquisas de mestrado: quatro aplicações destinadas às etapas de modelagem e/ou detalhamento de estruturas e duas aplicações que abordam tanto a etapa de modelagem como a de análise estrutural. Foram desenvolvidos também cinco projetos de pesquisa: um sistema para reestruturação do Modelador 3D (um sistema de modelagem estrutural desenvolvido pelo grupo), um sistema para orçamento de estruturas de concreto pré-moldado, uma aplicação para a automação de projetos elétricos, um sistema de geoprocessamento e um sistema para automação do projeto de placas de sinalização.

As pesquisas desenvolvidas têm trazido como resultados dissertações e sistemas de automação, mas, principalmente, têm despertado o interesse da comunidade científica por esta linha de pesquisa. Vale mencionar que esta atividade requer um esforço extra na formação de recursos humanos cujo perfil tem tido grande demanda no mercado atual. Neste sentido, as pesquisas nesta área têm cumprido um papel importante de qualificação e treinamento de profissionais e alunos em várias áreas do conhecimento como: a linguagem de programação C++, a POO, tecnologia CAD, interface gráfica, arquitetura de sistemas, padrões de projeto, bancos de dados, etc. Vários trabalhos deste grupo de pesquisa foram desenvolvidos em parceria com a Indústria Mineira e foram, direta ou indiretamente, aplicados na prática da engenharia. Os aplicativos resultantes trouxeram benefícios às empresas nas quais foram inseridos,

contribuindo para uma melhoria significativa da produtividade dos processos e alavancando sua competitividade de mercado.

As parcerias do grupo CADTEC com a indústria têm objetivos como a procura por caminhos e a criação de oportunidades para a comunidade acadêmica exercer o seu papel como propulsora do desenvolvimento de tecnologias no setor produtivo. Os benefícios da automação de processos numa indústria, como a minimização de erros, a redução de custos e a melhoria da qualidade do produto e da competitividade, vão consolidar a relação Universidade-Empresa. A contra partida será a influência dos aspectos práticos educativos desta relação no processo de ensino-aprendizagem.

Os trabalhos do grupo CADTEC foram desenvolvidos em espaços temporais diferentes, com objetivos gerais distintos, mas utilizando uma mesma tecnologia CAD. São sistemas desenvolvidos de forma independente para tipos de estruturas diferentes: estruturas prediais metálicas, torres metálicas, estruturas de concreto moldado *in loco,* estruturas de concreto pré-moldado e vasos de pressão. Apesar da independência existente entre os sistemas, fica clara a existência de um domínio de aplicação bem definido por um conjunto de dados e de funcionalidades semelhantes, presentes em cada um dos sistemas. É possível identificar ainda a presença de características específicas que dependem da estrutura tratada ou do conjunto de requisitos definidos para o sistema.

A aplicação da tecnologia CAD na automação do processo de modelagem de estruturas foi iniciada no grupo CADTEC através de uma parceria com a Empresa CODEME - Indútria de Estruturas Metálicas. Como resultados desta parceria foram gerados duas dissertações de mestrado e um sistema CAD, destinado à modelagem unifilar tridimensional de estruturas metálicas prediais denominado Modelador 3D (MALARD, 1998 e HÜNTER, 1998). Posteriormente, novas parcerias foram concretizadas com as empresas C.R. Gontijo e PREMO, com o objetivo de produzir sistemas CAD, respectivamente, para a automação da modelagem geométrica sólida de torres metálicas de transmissão de energia elétrica (MAGALHÃES, 2002) e para automação e detalhamento de estruturas de concreto pré-moldado (LEITE, 2002). Em ambos os casos, dissertações de mestrado foram também geradas.

Novos conceitos e novas abordagens foram também utilizados para o estudo de estruturas metálicas submetidas a resitência última, onde os conceitos de automação via tecnologias CAD/CAE foram aplicados (SOARES, 2006). A base de desenvolvimento de todos estes trabalhos é o modelo unifilar da estrutura. Além destes sistemas aplicados às estruturas reticuladas foi desenvolvido um sistema para modelagem, análise e pós-processamento aplicando a técnica de análise limite a vasos de pressão (BALABRAM, 2000).

São características comuns aos sistemas desenvolvidos no CADTEC:

- São sistemas orientados a objetos, desenvolvidos na linguagem C++, utilizando uma *Aplication Program Interface* (API), que possibilita a criação de aplicativos como bibliotecas de vínculo dinâmico, associadas a uma plataforma gráfica. Esta plataforma desempenha o papel de sistema anfitrião, carregando e gerenciando a aplicação durante uma seção de trabalho.

- São dependentes da plataforma gráfica. Utilizam amplamente os recursos gráficos desta plataforma: recursos de gerenciamento de dados e seu banco de dado proprietário;

- São aplicativos CAD, cuja missão é a automação da modelagem e/ou do detalhamento de estruturas reticuladas, com exceção de sistema CONRE (BALABRAM,2000) aplicado a vasos de pressão;

- Dados, controle e representação estão normalmente presentes dentro de cada abstração do objeto real.

Os trabalhos abordados por esta revisão bibliográfica são aplicados a dois domínios: o processo de projeto de estruturas reticuladas e POO aplicada ao Método dos Elementos finitos (MEF). Um domínio é formado por um conjunto de problemas correlatos os quais possuem características gerais, comuns a todos os problemas, e características específicas, que dependem de particularidades do problema tratado. Características gerais e específicas estão presentes em todo o domínio.

Para o projeto de estruturas reticuladas vários dados e decisões de projeto são independentes do tipo de estrutura, enquanto outros vão variar com: o material utilizado, a conformação geral dos elementos ou ainda com o tipo de execução e com a montagem da estrutura. Também para a POO aplicada ao MEF, algumas características são gerais, independentes do tipo de domínio analisado, enquanto outras dependem deste ou são específicas da técnica de análise implementada.

Dos trabalhos desenvolvidos pelo grupo CADTEC, foram selecionados para a pesquisa bibliográfica as dissertações citadas anteriormente além de um projeto de pesquisa aplicado à modelagem de estruturas reticuladas. Na área de POO aplicada ao MEF foram analisados cinco trabalhos: FEM_OBJ, um sistema orientado a objetos para análisse via elementos finitos; PZ, um ambiente orientado a objetos genérico, aplicado ao desenvolvimento de programas de elemento finitos; BRFEM, um *framework* para aplicações, destinado à análise via elementos finitos de campos eletromagnéticos; INSANE, um projeto de *software* livre desenvolvido para modelagem, análise e pós-processamento de estruturas reticuladas e o módulo Collapse do sistema WinCollapse (SOARES, 2006), que implementa a formulação para análise limite de pórticos planos via orientação a objetos.O estudo de sistemas de análise teve como objetivo principal: levantar as características gerais adotadas na abstração dos dados necessários a estes sistemas, procurando identificar uma base comum aos problemas abordados. Uma vez que a presente tese tem como objetivo propiciar o desenvolvimento de sistemas que façam a vinculação entre as diversas etapas do projeto de estruturas reticuladas, e como o MEF é um dos principais mecanismos utilizados pelos sistemas de análise disponíveis, torna-se importante produzir uma estrutura de classes compatível com os dados necessários a esta teoria. Foram estudados: a arquitetura geral dos sistemas através da organização dos dados em estruturas de classes, o relacionamento entre os objetos definidos, a representação dos processos desempenhados, a abstração das diferentes variáveis envolvidas no processo, os mecanismos de flexibilização de dados, etc. Garantir uma compatibilidade entre as representações geométricas e computacionais do modelo estrutural e dos modelos de análise, simplifica e agrega produtividade e eficiência ao processo de transferência de dados. Várias decisões de projeto, estruturas

de organização e mesmo de controle de fluxos adotados pelos sistemas analisados foram adaptadas e aplicadas ao domínio da presente pesquisa.

3.2 Modelador 3D (HÜNTER, 1998 e MALARD, 1998)

O primeiro sistema desenvolvido pelo grupo CADTEC foi resultado de duas pesquisas de mestrado: HÜNTER (1998) e MALARD (1998) desenvolvidas em parceria com a empresa CODEME de estruturas metálicas prediais. O Modelador 3D, como foi nomeado, é um sistema CAD que permite a modelagem de estruturas reticuladas através de um modelo unifilar. Trabalha-se com um modelo composto pelos elementos: barra, nó e superfície. O sistema realiza uma numeração automática dos elementos barra e nó, inseridos no modelo. A consistência do sistema é garantia pela vinculação entre elementos. O sistema permite ainda a aplicação de carregamentos nos elementos gerados.

Como os demais aplicativos desenvolvidos pelo grupo CADTEC, o Modelador 3D propõe uma estrutura de classes que visa utilizar, através de herança, as funcionalidades das classes nativas do sistema proprietário (*AcDbEntity* e *AcDbObject*[7]), a plataforma gráfica AutoCAD (FIGURA 3.1).

No Modelador 3D, as entidades do modelo unifilar são representadas por três classes principais. A classe **Nó** representa os nós ou pontos de encontro de barras. A classe **Barra** faz a abstração das barras, elementos lineares delimitados por um nó

[7] A classe *AcDbEntity* representa entidades que possuem representação gráfica, disponibilizando as funções que permitem gerar e manipular esta representação. A classe *AcDbObject* representa os objetos armazenados pelo banco de dados da plataforma, mas que não possuem uma representação gráfica.

inicial e um nó final. Já a classe **Face** modela as superfícies planas horizontais, que são planos delimitados por barras formando polígonos de três ou mais vértices, com uma direção de trabalho determinada.

Estas classes desempenham diversos papéis dentro do sistema, armazenando dados de geometria, dados de consistência e de vinculação, condições de contorno aplicadas, além de implementar a apresentação gráfica dos elementos.

As condições de contorno são representadas por carregamentos e restrições nodais. Para gerenciar os carregamentos foram criadas duas classes: a classe **Casos** que permite a criação e o gerenciamento de diferentes casos de carregamento, e a classe **CurCaso** que armazena o caso de carregamento corrente.

Para controle da numeração progressiva e independente dos elementos do modelo foi definida uma classe de configuração, **Enumeração,** que gerencia a numeração independente de barras, de nós e de faces. Esta classe também é responsável pela captura da numeração dos elementos excluídos do banco de dados, para que os números utilizados por eles possam ser reaproveitados na numeração de outros elementos criados posteriormente.

Alguns comandos de edição da plataforma gráfica podem ser aplicados normalmente sobre os elementos da estrutura, sendo gerenciados por classes de controle, derivadas de classes nativas da plataforma e que têm como função reagir às solicitações do usuário sobre as entidades representadas. Os objetos de controle são chamados *reactors*.

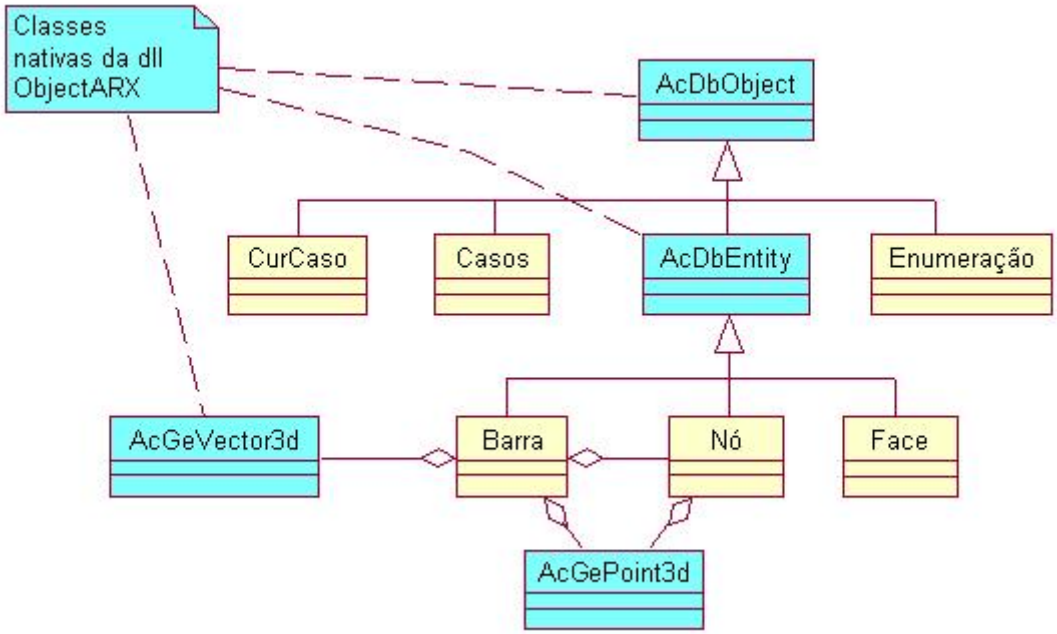

FIGURA 3.1– Estrutura de classes do Modelador3D

Análise comparativa com o *framework* proposto

A estrutura de classes definida neste modelador, assim como algumas funcionalidades básicas como gerenciamento de numeração de elementos e vinculação entre elementos foi reutilizada com novas implementações pela maioria dos sistemas CAD desenvolvidos pelo grupo CADTEC. No núcleo do REMFrame foram inseridas classes que realizam a abstração dos mesmos elementos tratados aqui, com exceção das superfícies. São as classes *Bar* (elemento barra), *Node* (elemento nó), *LoadCase* (caso de carregamento) e *Numbering* (enumerações). Apesar de representar os mesmos elementos do modelo estrutural tratados pelas classes do Modelador 3D, as novas classes implementadas possuem uma definição e um relacionamento completamente diferente das classes originais. Para a definição de sua estrutura e de seu relacionamento foram adotados critérios baseados em padrões e na análise das estruturas de classes propostas pelos sistemas CAE analisados nesta revisão. Uma das principais diferenças é a separação entre representação gráfica e dados, com a desvinculação da plataforma gráfica. As visualizações passam a ser implementadas a partir da classe abstrata *View*, mantendo uma independência em relação ao sistema gráfico utilizado.

Uma exposição mais completa da estrutura do *framework* REMFrame será apresentada no capítulo 5.

3.3 WINCONRE (BALABRAM 2000, FRANCO *et al*, 2006)

BALABRAM (2000) apresenta uma reformulação para o sistema CONRE[8] para a análise limite de cascas axi-simétricas, utilizando a POO aplicada ao MEF, englobando três etapas do processo, a saber: pré-processamento, análise e pós-processamento.

Foram desenvolvidos dois módulos: um sistema CAD, que funciona tanto como um modelador de cascas e gerador de malhas de elementos finitos, quanto como um pós-processador gráfico, e um sistema de análise orientado a objetos.

O subsistema de processamento gráfico disponibiliza comandos para a entrada de dados e o pós-processamento gráfico. Foram geradas duas estruturas de classes: uma para representar as entidades geométricas utilizadas e outra para representar as entidades e objetos do modelo discreto de elementos finitos.

---

[8] Originalmente o sistema CONRE foi desenvolvido na linguagem FORTRAN e apresentado por FRANCO J.R.Q. *Bounding Techniques in Shakedown and Ratchetting*. Tese de Doutorado, University of Leicestes, 1987

As classes geométricas (FIGURA 3.2) representam cascas primitivas axisimétricas, que permitem ao usuário reproduzir a geometria de vasos de pressão utilizados nas indústrias. São cascas cilíndricas (classe *CCylindric*), cônicas (classe *CCone*), toroidal (classe *CToroid)* e esféricas (classe *CSpheric*).

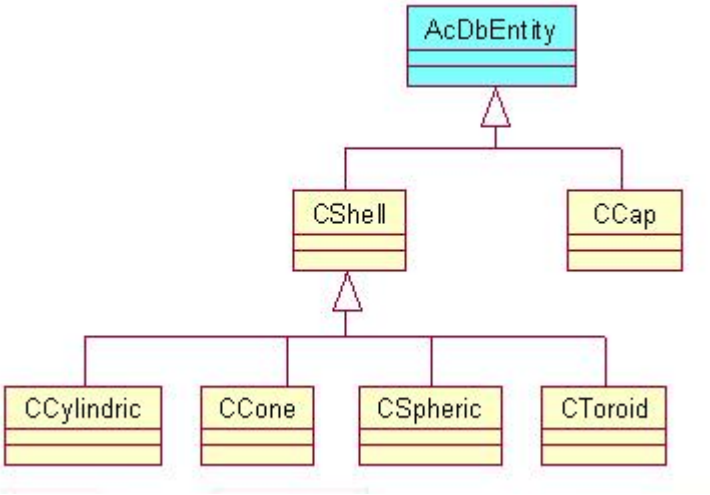

FIGURA 3.2 – Classes de representação geométrica do sistema CONRE

A classe *CShell*, abstrata, é a classe base para a definição das cascas primitivas. A classe *CToroid* permite suavizar a junção entre duas cascas e a classe *CCap* implementa a representação gráfica da tampa do vaso de pressão.

As entidades de representação gráfica dos objetos do modelo de elementos finitos (FIGURA 3.3) representam o elemento axi-simétrico de dois nós (*CElement*), os nós (*CNode*), as restrições (*CConstraint*) e as cargas aplicadas (*CLoad*) que podem ser a pressão interna (*CIntPressure*) e a carga de anel (*CRingLoad*), aplicada em toda a circunferência do vaso de pressão. A classe *CIsometricMaterial* armazena as propriedades do material utilizado, não possuindo uma representação gráfica.

O subsistema de análise realiza a análise de vasos de pressão via MEF, tendo sido implementado segundo a POO, tendo como base o trabalho apresentado por DUBOIS-PÈLERIN e ZIMMERMANN (1993) e estudado com mais detalhe no item 3.8 da presente tese. O sistema foi desenvolvido sobre as classes originais disponibilizadas pelo referido autor tendo sido implementadas alterações sobre algumas

classes e a definição de novas classes para representação de dados específicos da análise limite.

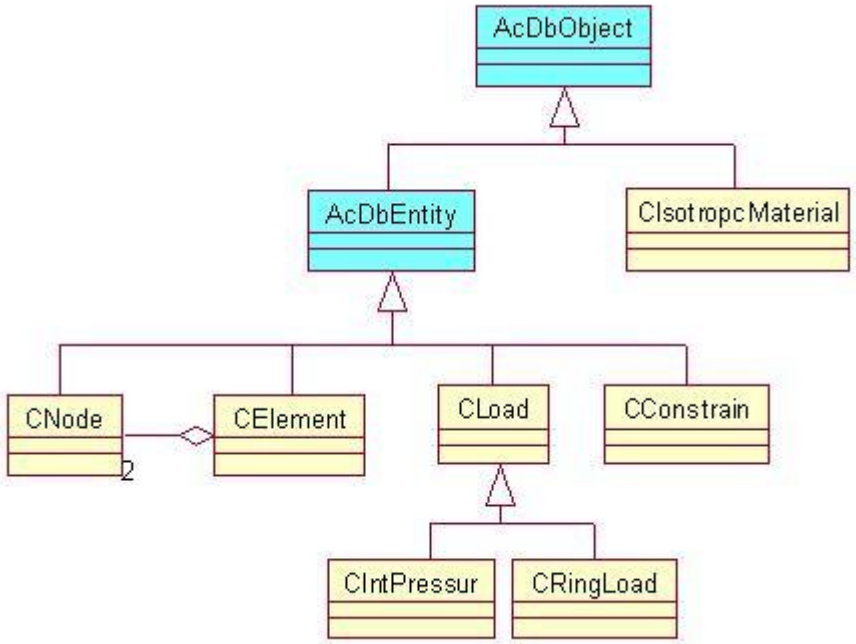

FIGURA 3.3 – Classes de representação de elementos do MEF do sistema CONRE

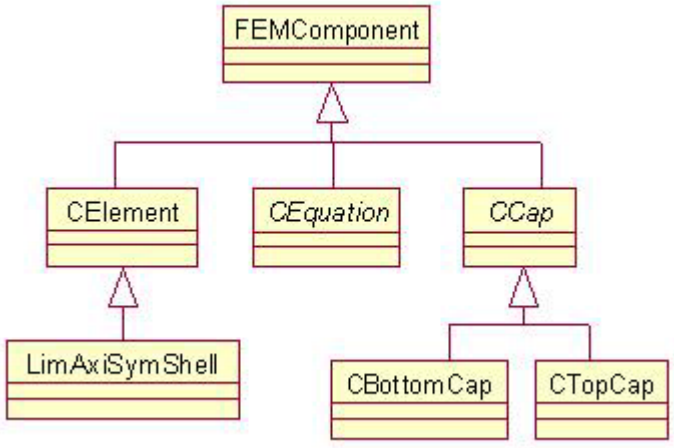

FIGURA 3.4 – Novas classes implementadas pelo subsistema de análise do CONRE na estrutura original proposta por DUBOIS-PÈLERIN e ZIMMMERMANN (1993)

A estrutura de classes original proposta por Zimmermann é apresentada na FIGURA 3.15. Esta estrutura não sofreu alterações quanto a sua organização geral, e as

classes implementadas por BALABRAM foram inseridas na hierarquia da classe *FEMComponent* (FIGURA 3.4). As classes implementadas foram: *LimAxiSymShell* que representa um elemento de dois nós para análise limite de cascas axi-simétricas de paredes finas, a classe *CEquation* e derivadas que representam equações de restrição cujo conjunto constitui o sistema de equações de restrição do problema de programação linear, a classe *CCap* e derivadas, responsável por definir tampas nas extremidades do vaso de pressão. A FIGURA 3.5 apresenta a geometria, a malha refinada de elementos finitos e a deformada para um vaso de pressão analisado através do sistema WINCONRE.

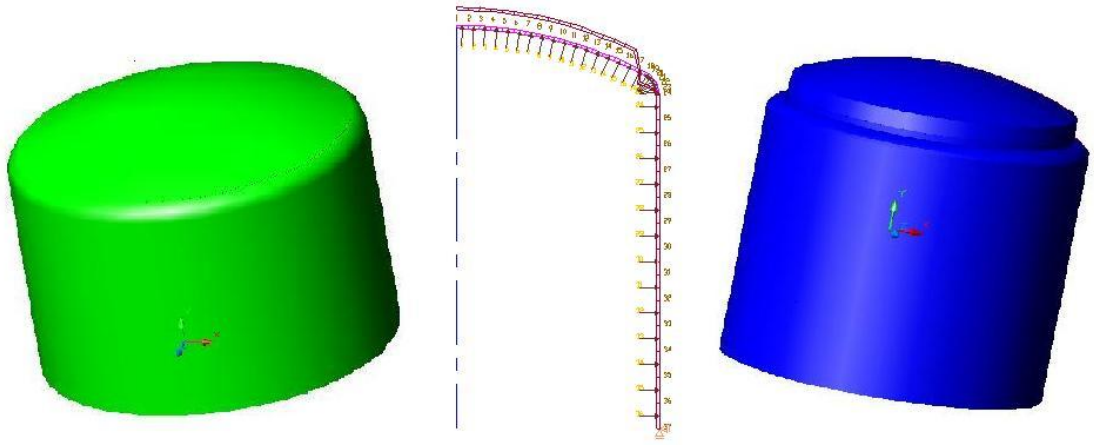

FIGURA 3.5 – Geometria, malha e deformada de vaso de pressão obtidos pelo sistema WINCONRE.

Análise comparativa com o *framework* proposto

Neste sistema vemos uma primeira implementação onde classes de representação gráfica (*CNode*, *CElement*, *CConstraint* e *CLoad*) implementam uma visualização para alguns objetos do sistema de análise. Estas classes possuem atributos que permitem definir a representação a ser gerada e métodos que implementam esta representação na plataforma utilizada. A filosofia de implementar dados e visualização em elementos diferentes, porém relacionados foi implementada no *framework* REMFrame através do padrão *Model-View-Controller*, abordado no item 4.2.4 desta tese. Entretanto na estrutura de classes do *framework* as classes *Node*, *Element*, *Load* e

Material fazem a representação dos dados dos elementos estruturais, ficando a visualização dos objetos de cada uma destas classes sob a responsabilidade de classes derivadas da classe *View*. A abstração de restrições nodais no *framework* não foi realizada por uma classe específica, como implementado por Balabram através da classe *CConstraint*. Os dados sobre restrições e liberações são então armazenados em classes contêiner[9] que reúnem todos os elementos de um modelo estrutural, podendo contar com visualizações independentes, implementadas por classes do tipo *View*.

De maneira semelhante à abordagem realizado por Balabram ao implementar uma estrutura de classes para a abstração da geometria dos elementos modelados através da classe base *Shell*, a geometria das seções transversais aplicadas aos elementos lineares tratados pelo *framework* é realizada pelas classes *Section, Shape*, e derivadas enquanto as classes *Line* e *Point*, realizam a abstração da geometria dos elemento lineares.

3.4 ArmaCAD (CARMO, 2001)

A terceira pesquisa realizada pelo grupo CADTEC teve como objetivo principal o desenvolvimento de um sistema CAD para detalhamento de estruturas de concreto convencional. Como resultado desta pesquisa foi desenvolvido o sistema ArmaCAD,

[9] No *framework* REMFrame as classes contêiner são aquelas responsáveis por armazenar um conjunto de informações ou dados correlacionados e que irão definir um modelo de um determinado projeto.

um aplicativo que realiza o lançamento automatizado do desenho de forma de vigas e de pilares. A aplicação não trabalha sobre um modelo estrutural, mas gera uma representação bidimensional de elementos isolados. A interface de usuário é realizada través de caixas de diálogo onde o usuário define as dimensões externas do elemento, a área de ferragens, número de camadas de ferragem, etc. O sistema não realiza o dimensionamento das peças nem das armações, mas posiciona o desenho das barras longitudinais nos desenhos de forma, tanto em vista quanto em corte, e o desenho dos estribos nas seções transversais. Na vista de vigas é inserida a representação das armações positiva e negativa e as posições dos elementos de apoio: pilares extremos e intermediários ou vigas extremas.

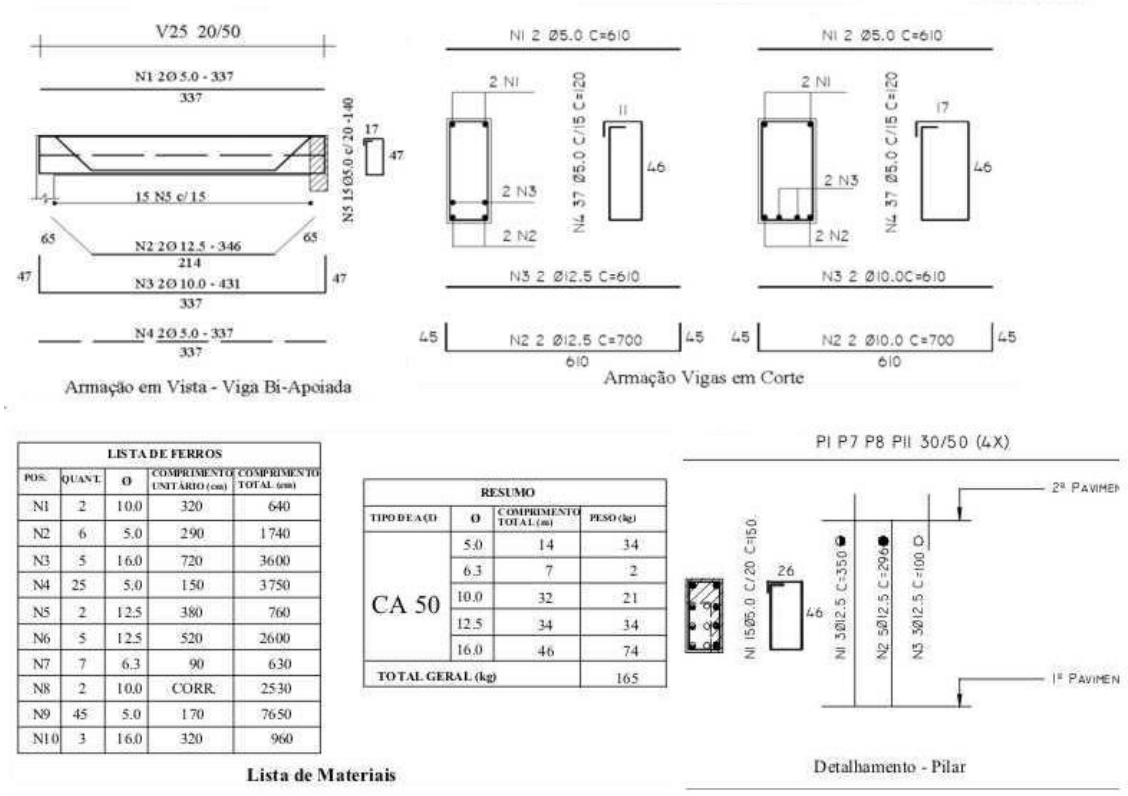

FIGURA 3.6 – Desenhos de forma e de armação gerados pelo sistema ArmaCAD.(figura extraída de CARMO, 2001)

Para representar os elementos de detalhamento foi definida a estrutura de classes representada na FIGURA 3.7. A classe **CCaixa** abstrai os dados que caracterizam um desenho de forma do elemento estrutural. A classe **CCorte** representa uma seção transversal da peça, e a classe **CVista** abstrai uma vista frontal do elemento estrutural. As classes **CArmacaoCorte** e **CArmacaoVista** encapsulam métodos e atributos para permitir a manipulação de dados e a representação dos elementos de armação dentro dos desenhos de forma. A classe **CLista** organiza e armazena os dados dos elementos do modelo dentro do banco de dados da plataforma gráfica utilizada.

A FIGURA 3.6 apresenta as caixas de diálogo implementadas pelo sistema ArmaCAD e a apresenta alguns desenhos de forma gerados pelo sistema.

Análise comparativa com o *framework* proposto

A representação de desenhos de detalhamento não foi abordada na atual versão do *framework* REMFrame, porém o estudo deste sistema foi importante para identificar a necesidade de se permitir uma representação independente destes desenhos para os elementos estruturais.

A estrutura proposta pelo *framework* permite que sistemas de detalhamento sejam implementados a partir do mesmo em futuras aplicações. Uma vez que dados e visualização são tratados de forma independente, permitindo a definição de novas classes de visualização nas aplicações desenvolvidas, a arquitetura proposta permite uma expansão de funcionalidade para a representação de detalhamento de elementos pela instanciação de novas classes concretas. Todos os dados referentes ao elemento estrutural podem ser armazenados pelas classes de dados (*Element* e derivadas, *Section*, *Shape* e derivadas) manipuladas pelo núcleo do *framework* e a abstração de elementos decorrentes do dimensionamento podem ser realizadas também através da definição de novas classes concretas. Assim, a estrutura do *framework* mostra que pode ser utilizada sem maiores alterações tanto para o desenvolvimento de sistemas de detalhamento, quanto para o desenvolvimento de aplicações que integrem estes sistemas a outros existentes no processo como pré-processadores e sistemas CAE.

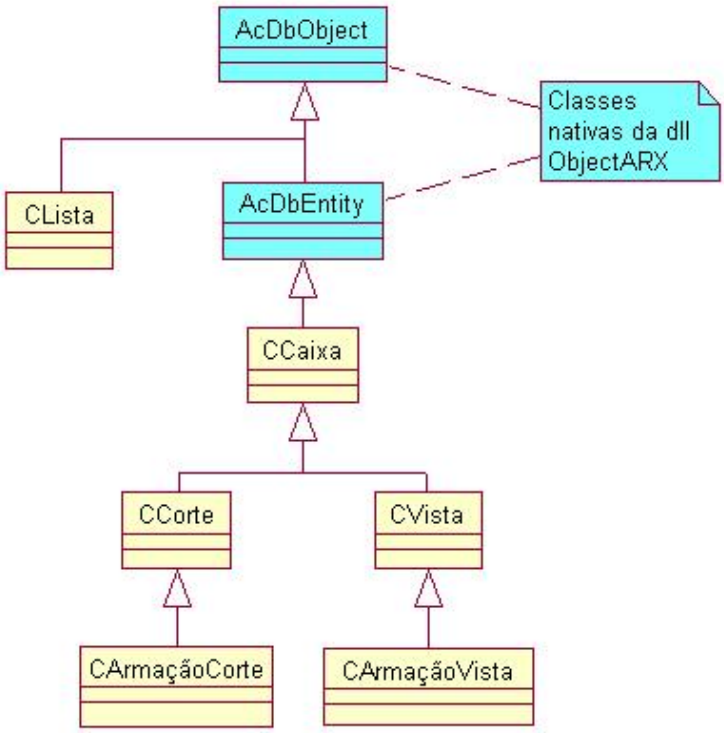

FIGURA 3.7 – Estrutura de Classes do ArmaCAD

3.5 TowerCAD (MAGALHÃES, 2002)

O quarto sistema, desenvolvido em parceria com a empresa C.R. Gontijo, foi denominado modelador TowerCAD e teve como meta a automação do detalhamento de torres de transmissão de energia elétrica, mais especificamente o detalhamento do tronco da torre e do *Stub*, uma peça da fundação de torres que realiza a ligação entre o bloco de fundação e o tronco da torre. Nesse modelador, os elementos da estrutura reticular recebem a atribuição de uma seção transversal, permitindo a representação do modelo estrutural através de um modelo geométrico sólido. Este modelo sólido armazena ainda os dados da representação unifilar da estrutura com numeração de

barras e de nós. O sistema também automatiza o detalhamento gráfico tridimensional dos stubs e de ligações de topo entre os elementos do troco da torre. Foram previstas ainda classes para o tratamento das ligações entre elementos da torre através de uma chapa de ligação.

Para executar o lançamento do tronco da torre o TowerCAD efetua a modelagem da estrutura a partir da leitura de um arquivo ASCII, oriundo de um sistema de análise e de dimensionamento de torres. Nesse arquivo são informados dados sobre os elementos do modelo: número total de barras e de nós, a vinculação entre barras e nós, as coordenadas nodais, os dados da seção aplicada a cada barra, uma codificação para cada barra que representa sua classificação (montante, treliça, etc.), sua posição relativa às faces da torre e a orientação da seção transversal.

Após a leitura dos dados, o modelo geométrico sólido é gerado. Cada elemento sólido representa uma barra do modelo unifilar, tendo um elemento nó em cada extremidade, semelhante ao modelo tratado pelo Modelador3D (FIGURA 3.8).

FIGURA 3.8– Elementos e ligações modeladas pelo TowerCAD (figura extraída de MAGALHÃES, 2002)

A estrutura de classes desenvolvida para a abstração de torres de transmissão elétrica é apresentada na FIGURA 3.9. As classes **Barra** e **Nó** têm funções similares às classes de mesmo nome, definidas pelo Modelador 3D. Para a modelagem geométrica sólida foram definidas três novas classes: **Perfil**, **Ligação** e **Chapa**. Estas classes

realizam a abstração e a manipulação de dados dos elementos: seção transversal, ligação de topo entre elementos e ligação de chapa, respectivamente. Estas classes também são responsáveis por definir a representação gráfica dos elementos. A classe **Perfil** utiliza superfícies e sólidos da plataforma gráfica para definir uma seção transversal e criar uma representação sólida do elemento estrutural pela extrusão desta seção ao longo de um elemento do tipo **Barra**. As ligações entre elementos do tipo **Barra** são modeladas pela classe **Ligação**. A partir de parâmetros de definição do tipo de ligação adotada entre os elementos do tronco da torre, esta classe efetua tanto as verificações de norma para o posicionamento dos elementos de ligação, cantoneiras e parafusos, quanto a representação dos elementos envolvidos. Para a ligação de chapa foi definida a classe **Chapa**, que tem atribuições semelhantes às da classe **Ligação**, porém tratando da ligação de duas ou mais barras através do dimensionamento e da representação de um terceiro elemento estrutural, isto é: a chapa de ligação.

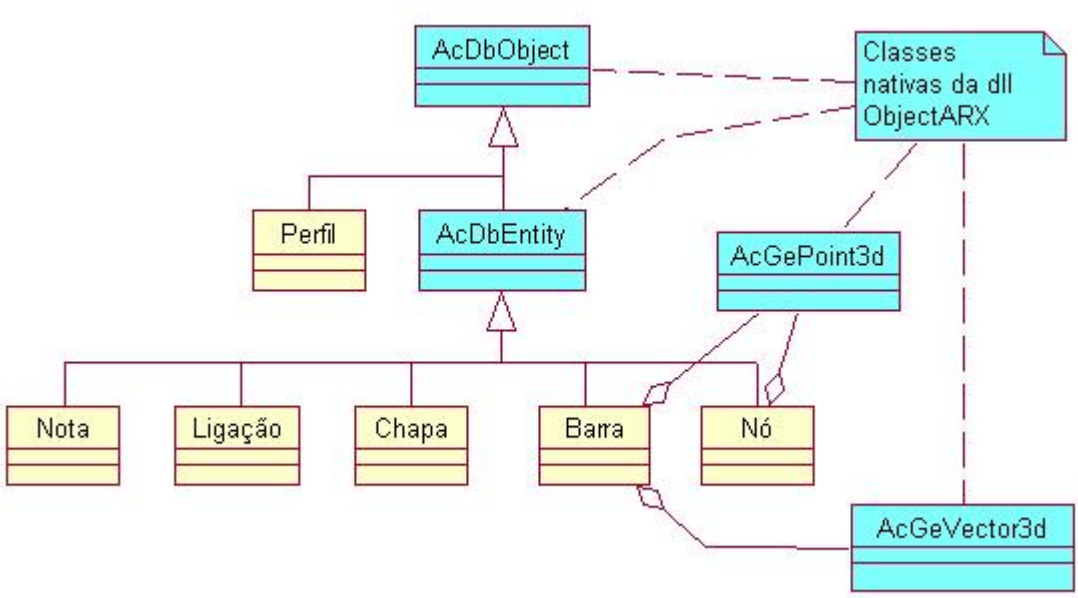

FIGURA 3.9 – Estrutura de classes do sistema TowerCAD

Análise comparativa com o *framework* proposto

No *framework*, uma seção transversal pode ser composta por uma ou mais formas geométricas. A seção composta é representada pela classe *Section*, enquanto as

formas geométricas são representadas pela classe *Shape* e derivadas. Enquanto Magalhães propõe uma abstração conjunta de dados e representação gráfica em um só elemento (*Perfil*), no *framework* a representação dos dados da seção transversal aplicada aos elementos é separada da sua representação gráfica (classes *View*). O tratamento de propriedades geométricas para as diferentes classes *Shape* é realizado pela classe *GeomPropHandler,* e derivadas. As classes abstratas *ElementConection*, *Conection* e *ConectionRule* foram propostas para o tratamento dos dados de ligações entre elementos no *framework,* devendo ser instanciadas por aplicações que realizem o detalhamento das ligações.

3.6 PREMOLD (LEITE, 2002)

O PREMOLD foi o quinto modelador estrutural desenvolvido. Este modelador teve como meta a automação da modelagem e do detalhamento de estruturas de concreto pré-moldado. Na implementação deste modelador foi realizada uma re-estruturação das classes de representação do modelo unifilar apresentadas pelo Modelador 3D com o objetivo de absorver neste novo modelador as soluções ali implementadas de vinculação, consistência e numeração de elementos. Sobre a estrutura de classes original do Modelador 3D foram realizadas as seguintes implementações: atualização de versão da plataforma gráfica utilizada, uma re-estruturação da arquitetura do sistema, uma redefinição das classes originais – atributos, métodos e relacionamentos.

O detalhamento das ligações entre os elementos pré-moldados foi realizado tanto pela modelagem sólida das ligações quanto pela geração dos desenhos de forma dos elementos. A modelagem sólida das ligações é realizada com inserção e dimensionamento inicial de elementos de ligações como consoles e dentes de Gerber[10]. A modelagem geométrica sólida foi realizada de maneira semelhante à abordagem feita pelo TowerCAD.

Nesta aplicação, o lançamento da estrutura é realizado pelo usuário através de diversos comandos que automatizam a inserção de elementos do tipo pilares, vigas e painéis horizontais. A escolha da seção transversal destes elementos é realizada em tempo de execução a partir de uma biblioteca de seções inserida no aplicativo. Foram previstas seções retangulares parametrizadas para pilares, vigas e painéis de piso, permitindo a inserção de elementos com dimensões variáveis além daquelas previstas pela biblioteca de seções. A definição e inserção de novas seções não foi implementada nesta versão.

Além da modelagem sólida, foi implementada a geração dos desenhos de forma dos elementos inseridos no modelo, com automatização dos desenhos de vista e seção transversal de cada elemento (FIGURA 3.11). Os desenhos são obtidos a partir dos dados armazenados na biblioteca de seções utilizadas. Como o sistema faz a modelagem sólida, mas não contempla o dimensionamento dos elementos, não foram inseridos desenhos de armadura.

---

[10] Dentes de concreto para apoio em viga com recorte, cujo emprego é bastante comum na pré-moldagem. (EL DEBS, 2000)

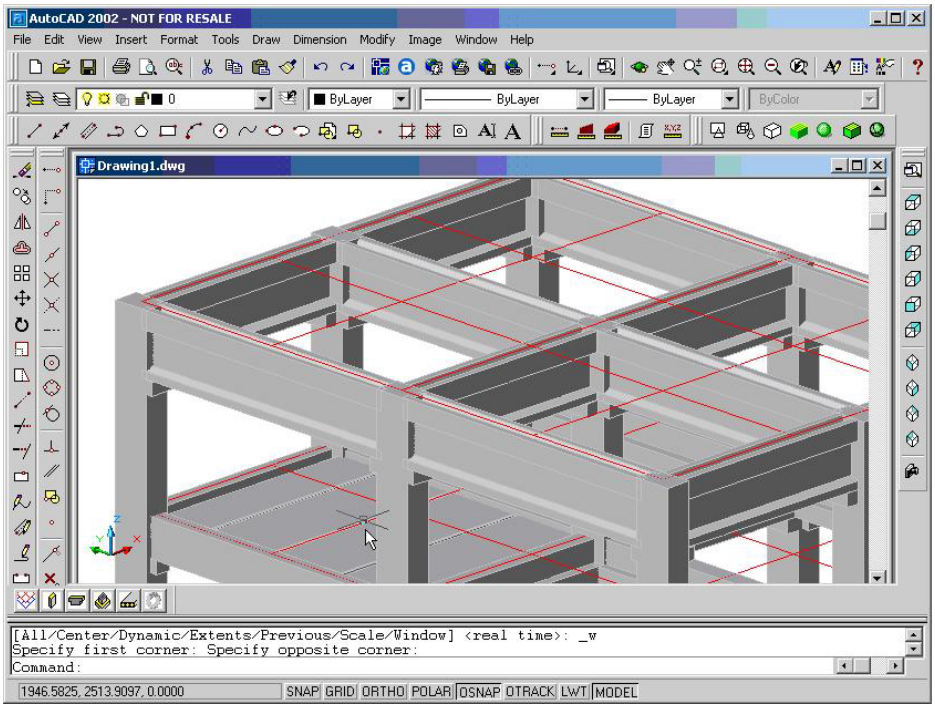

FIGURA 3.10 - Modelagem sólida pelo PREMOLD (figura extraída de LEITE, 2002)

O projeto do novo sistema foi desenhado procurando representar os diversos elementos que constituem uma estrutura de concreto pré-moldado. A estrutura de classes construída teve como foco os preceitos de herança e de polimorfismo da orientação a objetos.

Todos os atributos de classes relacionados às condições de contorno, à numeração, à consistência e à validação do modelo foram implementados a partir do estudo de reestruturação realizado sobre o Modelador 3D. Como resultado desta reestruturação, foi apresentada uma nova estrutura de classes (FIGURA 3.12), onde as classes do Modelador3D são reescritas e inseridas dentro de uma hierarquia que engloba tanto a modelagem unifilar quanto a modelagem geométrica sólida de elementos estruturais e de elementos de ligações.

A classe **Elem_Estrutural** reúne as características comuns a todos os elementos estruturais modelados e dela foram derivadas classes para representação dos elementos prismáticos (**Elem_Linear**) e plano (**Elem_Plano**). A classe **Bloco** foi definida para

representar um nó de apoio onde é inserido um bloco de fundação que é pré-dimensionado durante sua inserção segundo a norma NBR9062 [11]

A classe **Consolo** realiza a representação dos elementos de apoio de vigas anexados aos pilares pré-moldados. Como podem existir consolos com comprimentos variáveis podendo chegar a um metro em algumas situações, esta classe foi derivada da classe **Elem_Linear**, assim como as classes **Viga** e **Pilar** que representam os elementos estruturais de mesmo nome. A classe **Painel** representa elementos planos tanto de vedação vertical quanto elementos de piso, porém apenas os comandos de inserção de painéis de piso foram implementados na versão apresentada.

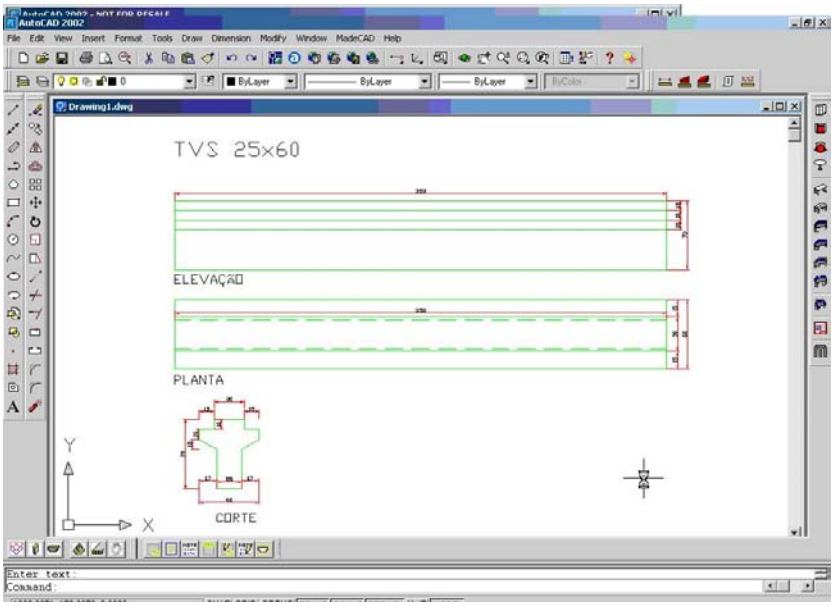

FIGURA 3.11 – Desenho de forma gerado pelo PREMOLD

(figura extraída de LEITE, 2002)

[11] NBR 9062 -Projeto e execução de estruturas de concreto pré-moldado – Procedimento (orig. NB949ABNT – Associação Brasileira de Normas Técnicas, Norma Técnica Brasileira,), 1985.

Durante a inserção dos elementos, o tratamento das ligações é feito automaticamente considerando: o dimensionamento geométrico e a inserção de consoles em pilares, os recortes necessários em vigas, considerando inclusive as folgas definidas pelo usuário, o posicionamento relativo entre elementos, o dimensionamento geométrico de blocos de fundação e de painéis de piso.

Os comandos de criação de elementos estruturais permitem a inserção conjunta de vários elementos de um mesmo tipo com variação do ponto de inserção da seção transversal aplicada e o eixo do elemento do modelo unifilar, rotação da seção em torno deste eixo e automatização do posicionamento relativo entre elementos que apresentam dentes ou encaixes especiais.

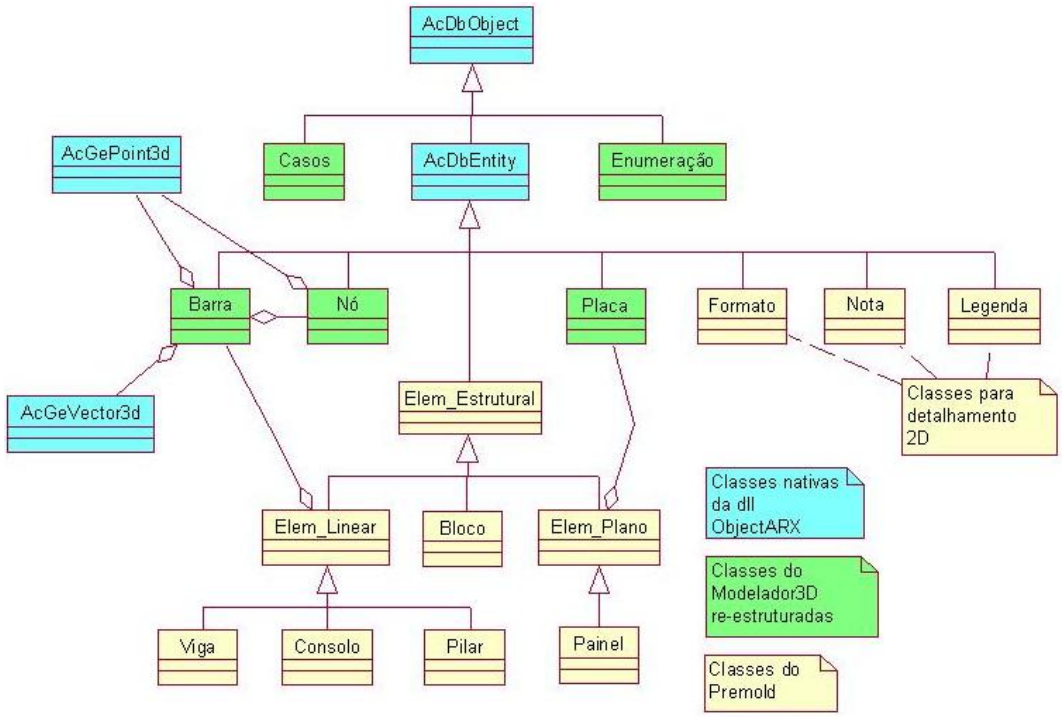

FIGURA 3.12 – Estrutura de classes do Modelador PREMOLD

Análise comparativa com o *framework* proposto

Algumas funcionalidades deste sistema foram implementadas na aplicação instanciada através do *framework* e que recebeu o mesmo nome, além da aplicação STEEMOLD que faz o lançamento de uma estrutura metálica predial. Foram

reimplementados a utilização de uma biblioteca de padrões, porém agora escolhida e carregada em tempo de execução. Uma rotina para definição de seções genéricas, implementada no núcleo do *framework*, permite agora a definição de qualquer seção composta em tempo de execução, sua inserção no modelo e sua persistência em bibliotecas de seções. O lançamento automatizado dos elementos também foi implementado com a inovação de permitir a inserção de elementos tridimensionais em qualquer direção, o que não era feito pelo PREMOLD original. É permitida, ainda, a definição e aplicação de materiais diferenciados e a aplicação de condições de contorno sobre a estrutura modelada. Apenas os desenhos de detalhamento e o tratamento de ligações não foram implementados, mas como mencionado anteriormente, estas funcionalidades podem ser implementadas em aplicações futuras ou em novas versões das aplicações PREMOLD e STEELMOLD.

3.7 Reestruturação dos Modeladores Modelador3D, TowerCAD, PREMOLD - 2002

No ano de 2002, após o desenvolvimento dos sistemas até aqui abordados, o grupo CADTEC se empenhou em um novo projeto de pesquisa cuja meta era o desenvolvimento de um novo sistema para modelagem de torres de transmissão. Esta nova versão reuniria em um sistema a funcionalidade do Modelador 3D, permitindo o lançamento interativo da estrutura, o modelo geométrico sólido e o detalhamento proposto pelo TowerCAD. Os conceitos de orientação a objetos foram reavaliados na definição de uma nova estrutura de classes, mais detalhada e voltada para a modelagem sólida.

Nesta mesma época iniciava-se a presente pesquisa, avaliando-se a aplicabilidade da nova estrutura, então em desenvolvimento, ao modelador PREMOLD.

A nova estrutura mantinha a base original adotada pelo Modelador3D e conseqüentemente pelo PREMOLD. A esta base foram inseridas classes genéricas para representar uma seção transversal do elemento estrutural e uma ligação entre elementos (FIGURA 3.13). No PREMOLD não existem classes para seções ou ligações. Os dados e procedimentos para inserção de seções transversais são abstraídos por atributos e métodos das classes derivadas da classe **Elem_Estrutural**. As ligações são tratadas através de operações booleanas entre os sólidos dos elementos relacionados a um nó.

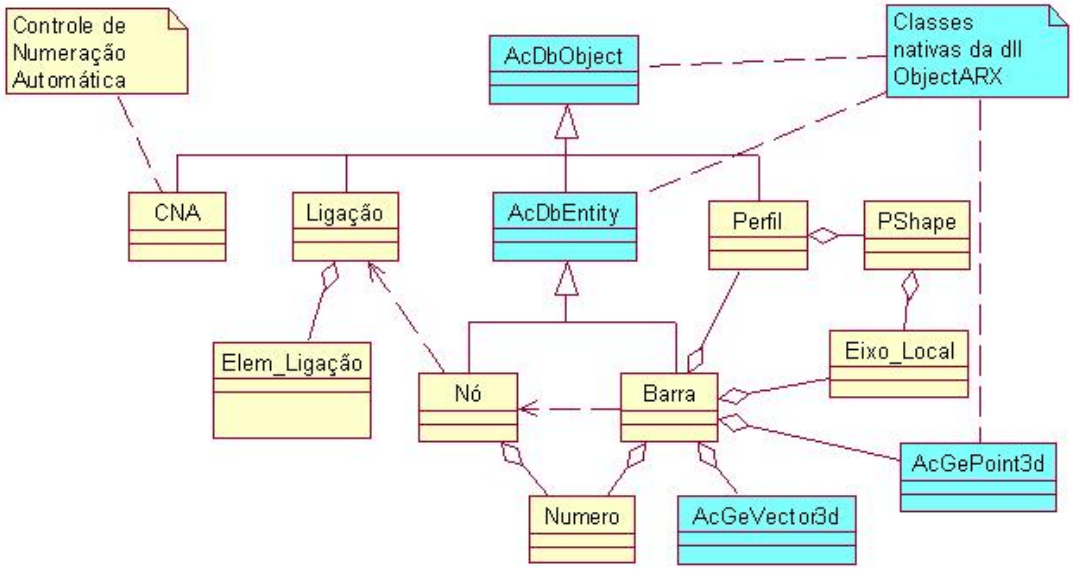

FIGURA 3.13 – Proposta inicial de classes do novo Modelador 3D

A partir desta estrutura básica, verificou-se que o desenvolvimento dos dois sistemas passaria a ser realizado em paralelo com definição de classes de perfis, de ligações de elementos, etc., para cada aplicação, devido às diferenças existentes entre as estruturas originais (FIGURA 3.14). Assim, a partir das classes apresentadas passou-se a ter uma ramificação em cada ponto onde as estruturas diferiam entre si. A implementação dos dois sistemas em paralelo acarretaria algumas conseqüências:

- As alterações inseridas no núcleo de um sistema tinham de ser copiadas para o outro para garantir a unicidade da base comum;

- Considerando-se o aumento de complexidade dos sistemas com o decorrer dos desenvolvimentos, a administração desta duplicidade de informações se tornaria cada vez dispendiosa;

- A arquitetura do sistema continuava muito dependente de herança, o que dificultava sua utilização por outros sistemas;

- A base comum continuava pequena se comparada às implementações específicas, necessárias para a geração de sistemas para estruturas diferentes;

- Os sistemas resultantes continuariam dependentes de uma plataforma específica.

A avaliação destes fatores permitiu a identificação de algumas questões importantes que precisavam ser respondidas antes de prosseguir no desenvolvimento destas aplicações:

- Como tornar este núcleo mais representativo para o desenvolvimento de outras aplicações para os diferentes tipos de estruturas reticuladas?

- Qual a melhor estrutura de classes para possibilitar seu reaproveitamento de forma mais direta no desenvolvimento dos diferentes sistemas?

- Como permitir o reaproveitamento de soluções adotadas para problemas mais específicos sem ter de utilizar todo o núcleo proposto?

- Como possibilitar o uso de outras plataformas gráficas no desenvolvimento destes sistemas?

- Qual a melhor representação dos dados da estrutura reticular pensando-se em uma futura vinculação destes sistemas gráficos a sistemas CAE para a análise, o dimensionamento e o detalhamento das estruturas, evitando redefinir estas etapas em um novo sistema, mas utilizando sistemas existentes?

- Como permitir a vinculação a estes diferentes sistemas CAE?

- Como flexibilizar a escolha destes sistemas CAE?

A busca por respostas a estes questionamentos definiu os objetivos específicos, a hipótese, os procedimentos, os instrumentos e as técnicas adotadas na presente tese, além de ter gerado a necessidade de estudar a arquitetura e as soluções adotadas em diferentes sistemas de MEF desenvolvido com o uso da POO. Os sistemas estudados foram escolhidos pela acessibilidade aos dados e informações necessárias (FEM_OBJ e PZ, itens 3.8 e 3.9) e pela similaridade de técnicas utilizadas (BRFEM, item 3.10) ou domínio de aplicação (INSANE, item 3.11).

O novo modelador apresentado neste item continuou a ser desenvolvido pelo grupo CADTEC paralelamente a presente pesquisa e sua versão mais atual foi apresentada por SOARES (2006), sendo abordada no item. 3.12.

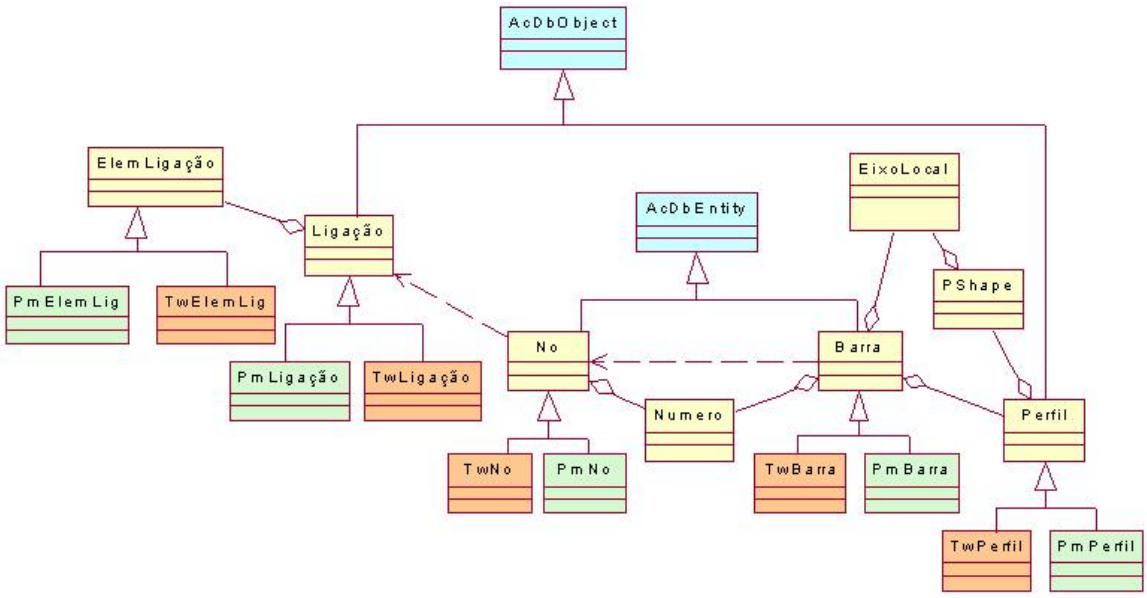

FIGURA 3.14 – Proposta de estrutura de classes para os novos modeladores TowerCAD e PREMOLD

Análise comparativa com o *framework* proposto

A abstração de dados realizada pelas classes *Section*, *Shape*, *ConectionElement* e *Conection* do *framework* foi baseada na definição das classes PShape, Perfil, ElemLigação e Ligação definidas por esse projeto de pesquisa observando-se, porém, a separação entre dados e visualização adotada no *framework* e a aplicação de padrões de

projeto para modelar os relacionamentos e comportamentos das estruturas em que são inseridas.

3.8 Sistema FEM_OBJ (ZIMMERMAN e*t al*, 1992, DUBOIS-PÈLERIN *et al*, 1992, DUBOIS-PÈLERIN e ZIMMERMAN,1993)

Os três artigos: ZIMMERMAN e*t al*, (1992), DUBOIS-PÈLERIN *et al*, (1992) e DUBOIS-PÈLERIN e ZIMMERMAN (1993) apresentam uma arquitetura orientada a objetos de um sistema para análise via elementos finitos. Os princípios fundamentais da aplicação da orientação a objetos aplicada ao MEF é apresentado por ZIMMERMAN e*t al* (1992). DUBOIS-PÈLERIN *et al* (1992) descreve um sistema protótipo escrito na linguagem Smalltalk com o uso da orientação a objetos Em busca de obter uma maior eficiência numérica para o sistema DUBOIS-PÈLERIN e ZIMMERMAN (1993) apresentam a arquitetura de classes do sistema transportado para a linguagem C++. A FIGURA 3.15 apresenta a estrutura de classes deste último.

As classes *Domain, Dof* e *FEMComponent* implementam objetos específicos da metodologia de elementos finitos. Um objeto da classe *Dof* gerencia as variáveis cinemáticas, o número de equações e as condições de contorno. A classe *Domain* (FIGURA 3.16) implementa a malha de elementos finitos. Seus atributos incluem elementos, nós, dicionários de materiais e o sistema linear. A classe *FEMComponent* é a classe base para classes comuns a todas as formulações de MEF através da POO: *Element, Node, Material* e *Load.* Outras classes derivadas de *FEMComponent* fazem o tratamento de dados especificamente para análises dinâmicas: *LoadTime Function Time Integration Scheme*(e derivadas) e *TimeStep*.

A classe *Element* representa características de elementos genéricos como identificação de nós, de material e de cargas aplicadas, número de pontos de Gauss, matriz de rigidez, matriz constitutiva, etc. A classe *Node* representa um nó do modelo, sendo responsável por gerenciar sua posição, reunir os carregamentos nodais e gerenciar

seus graus de liberdade, tendo como atributos as coordenadas nodais, dicionários de graus de liberdade, vetor de carregamentos nodais, etc. A classe *Load* e classes derivadas implementam todos os carregamentos aplicados aos elementos do modelo. A classe *Material* gerencia as propriedades físicas dos elementos.

As demais classes implementadas são: classes *GaussPoint* e *Polynomial* que implementam métodos para integração numérica; classe *Linear System* e derivadas implementam os solvers do sistema; as classes: *Dictionary, FloatArray, IntArray, List, String, Matrix*, etc., são classes contêineres, organizadas em uma biblioteca de classes.

A classe *FileReader* implementa métodos que permitem o acesso a arquivos de dados, a busca por tipos específicos de dados e a leitura dos mesmos.

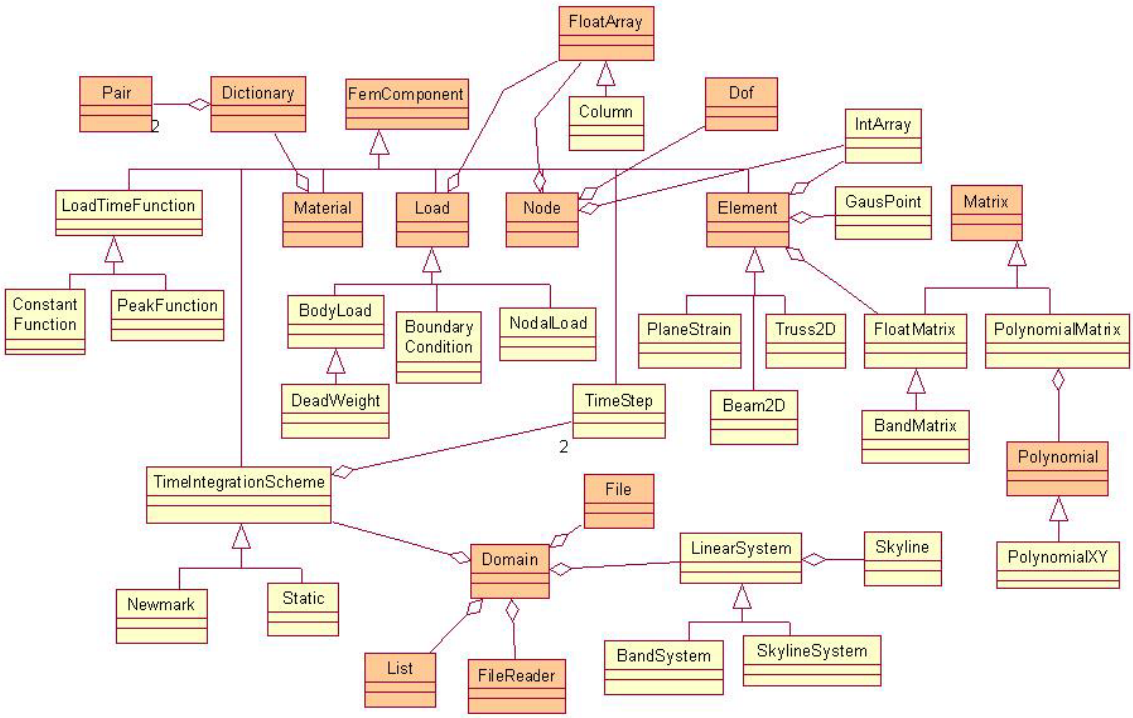

FIGURA 3.15 – Classes do sistema FEM_OBJ

Análise comparativa com o *framework* proposto

A estrutura de classes de dados do *framework* foi baseada inicialmente na estrutura do sistema FEM_OBJ, porém sem abordar as funcionalidades específicas do MEF, mas tratando dados do modelo estrutural essenciais para este método. Podemos traçar uma correlação entre as classes das duas arquiteturas da seguinte forma: a classe base *Element* do *framework* define uma estrutura semelhante a da classe *FEMComponent*, enquanto a classe *Bar* do *framework* aproxima-se mais da classe *Element* do FEM_OBJ. No *framework Load* e *Material* não são derivadas de *Element*. *Load* participa da hierarquia das classes *Subject* e *Componente*, da mesma forma que a classe *Element,* enquanto *Material* é derivada apenas de *Component.*

Assim como nos sistemas descritos nos próximos itens está presente neste sistema uma classe contêiner *Domain* que reúne os objetos elementos, nós, dicionários de materiais etc. de uma malha elementos finitos. Para reunir os dados dos elementos tratados pelo *framework* foi implementada a classe base *Model.* Esta classe reúne objetos do tipo *Bar* e *Node*, porém o dicionário de materiais foi inserido em uma outra classe contêiner, a classe *Project,* que reúne diversos objetos do tipo *Model*. A definição da classe *Project* foi baseada nos sistemas abordados nos próximos itens.

De maneira semelhante ao sistema FEM_OBJ, foram definidas algumas classes para manipulação de dados geométricos, organizadas em uma biblioteca de classes independente (componente *MathClasses*, item 5.1.1).

Uma classe base, que define uma interface para acesso a diferentes tipos de documentos para a persistência de dados, foi implementada (*Filler*) da qual a classe *FileFiller*, para o acesso a arquivos de dados, foi derivada. A interface da classe *Filler* e a implementação da classe *FileFiller* foi baseada na classe *FileReader* do sistema FEM_OBJ.

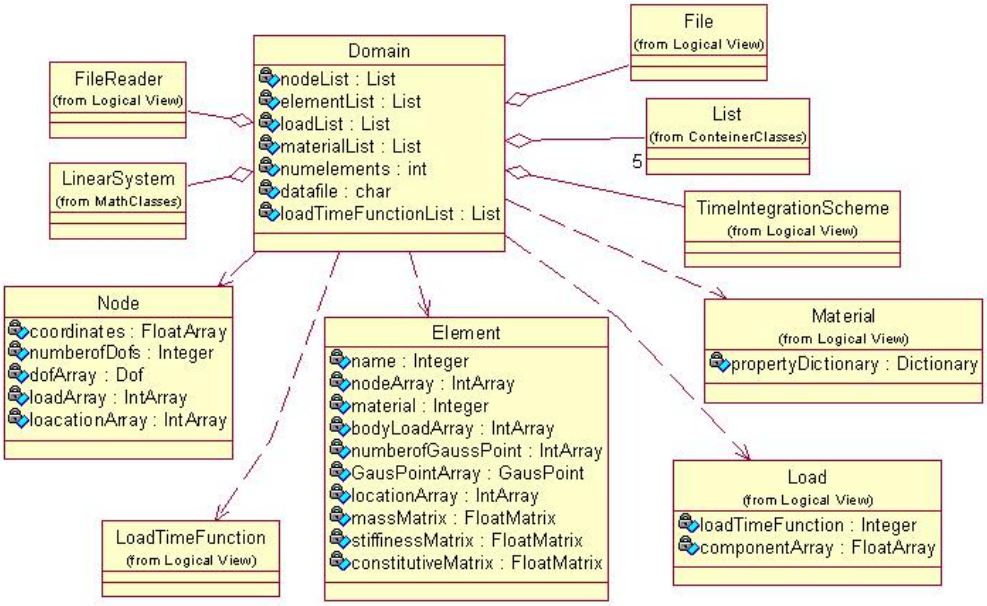

FIGURA 3.16 – Estrutura e das classes de domínio do sistema FEM_OBJ

3.9 PZ (DEVLOO, 1997, DEVLOO, 1999, PZ, 2005, *on line*)

DEVLOO (1999) apresenta a filosofia envolvida no desenvolvimento de ferramentas orientadas a objetos em três ambientes independentes: um ambiente para álgebra linear (TMatrix), um ambiente para desenvolvimento de programas de elementos finitos (PZ) e um ambiente para desenvolvimento de sistemas com o uso de programação paralela (OOPAR). O autor define um ambiente como “um conjunto

independente de classes que implementa uma determinada tarefa, (...) usualmente vinculado a uma filosofia de desenvolvimento." O ambiente de álgebra linear, TMatrix, é extensível para a inclusão de novas formas de armazenamento de matrizes, o ambiente de elemento finitos, PZ[12], separa o mapeamento geométrico da definição do espaço de interpolação e da definição do sistema de equações diferenciais parciais e o ambiente paralelo implementa conceitos de objetos de dados distribuídos e de tarefas.

O objeto de estudo inicial desta pesquisa foi o ambiente PZ descrito a seguir, porém, a filosofia adotada no ambiente OOPAR foi aqui também abordada porque esta, de alguma forma influenciou decisões de desenvolvimento da presente pesquisa, apesar desta última não estar relacionada à programação paralela.

Nesta seção, devido à complexidade do sistema PZ, a análise comparativa entre este e o *framework,* REMFrame, será apresentada ao longo da síntese.

3.9.1 O ambiente PZ

Segundo o autor, o PZ é um ambiente orientado a objetos genérico e extensível, aplicado ao desenvolvimento de programas de elemento finitos especializados para domínios específicos.

O ambiente PZ oferece estruturas separadas de classes para a aproximação geométrica, a definição da interpolação e para a definição da equação diferencial.

---

[12] http://www.fec.unicamp.br/~phil/

Neste ambiente, o conjunto de elementos e nós forma uma grade. Há uma grade de elementos para a representação do mapa geométrico e uma grade de elementos que definem as funções de forma e regras de integração relacionadas.

Classses de geometria do domínio:

A classe *TGeoGrid* é uma classe contêiner que armazena um conjunto de nós geométricos (*TGeoNod*), elementos geométricos (*TGeoEl*) e nós de contorno (*TGeoNodBc*). A malha geométrica contém a árvore completa dos elementos refinados.

Os elementos do tipo *TGeoEl* definem um mapeamento entre o elemento e sua configuração deformada. Cada elemento geométrico possui um identificador do material a ele associado. São definidos métodos para refinamento dos elementos geométricos e para a criação dos elementos computacionais a partir dos elementos geométricos. A classe *TGeoNod* define um ponto no espaço Euclidiano tridimensional e associa um identificador único ao mesmo.

A separação do mapeamento geométrico da interpolação permite a derivação de classes dos elementos geométricos, sem afetar o espaço de interpolação.

Análise comparativa com o *framework* proposto

No *framework,* REMFrame, os dados geométricos representados por objetos das classes *Bar* e *Node* são armazenados na classe contêiner *GeometricModel*, derivada da classe *Model*. Um elemento geométrico *Bar* do modelo *GeometricModel* é a representação de um elemento lançado pelo projeto estrutural (uma viga, um pilar, uma barra de treliça, um elemento de contraventamento, etc.). Um objeto do tipo *Bar* também armazena um indicador do material (classe *Material*) associado a ele. No REMFrame, são materiais o meio constitutivo do qual o elemento é construído, representado pelas suas propriedades físicas (classe *PropertyMap*). Neste caso utiliza-se o nome dado ao material.

No REMFrame são definidos métodos de refinamento do elemento geométrico para geração de um modelo refinado, neste caso não um modelo de elemento computacionais, mas uma malha refinada de elementos geométricos destinada a alimentação dos sistemas de análise utilizados no processo de projeto.

Na arquitetura do REMFrame a classe *Point* que realiza a representação de um ponto no espaço Euclidiano tridimensional, enquanto a classe *Node* representa um nó de extremidade ou intermediário do elemento estrutural, possuindo um objeto do tipo *Point* e uma numeração única para uma dada estrutura. Um objeto *Node* armazena em uma lista encadeada os identificadores dos objetos *Bar* a ele vinculados. Da mesma forma um objeto *Bar* possui uma lista encadeada os identificadores dos seus nós de extremidade e intermediários.

Classes de elementos computacionais do ambiente PZ:

A classe *TCompGrid* é a classe contêiner dos elementos computacionais e armazena listas de elementos computacionais (*TCompEl*), de graus de liberdade (*TDofNod*), de objetos materiais (*TMaterial*), de condições de contorno (*TBoundCond*); de condições de contorno nodais (*TDofNodBc*).

Uma malha computacional é sempre derivada de uma malha geométrica e várias malhas computacionais podem ser derivadas de uma única malha geométrica. Os elementos computacionais podem ser referenciados pelos elementos geométricos pela inclusão de um ponteiro para um elemento computacional em cada elemento geométrico, permitindo que cada elemento geométrico aponte para um elemento computacional por vez.

A classe *TCompEl* é uma classe abstrata. O elemento computacional define as funções de interpolação e contém o método para integrar a matrix de rigidez do elemento. A classe *TDofNod* implementa os graus de liberdade do nó e contém uma variável que indica o número de elemento a ele associado. O elemento computacional não armazena as informações sobre a simulação desenvolvida. A classe *TMaterial* é uma classe abstrata que define o problema físico e suas características incluem tanto o

cálculo das contribuições de rigidez quanto o pós-processamento. Segundo o autor a separação entre a definição do espaço de interpolação e a definição da equação diferencial que está sendo aproximada, permite que, uma nova tecnologia introduzida no nível geométrico, ou no nível computacional, seja automaticamente válida para todos os problemas físicos modelados pela classe *TMaterial*.

A classe *TBndCond* armazena os dados necessários para definir uma condição de contorno. A classe *TSuperEl* é derivada de ambos *TCompGr* e *TCompEl*. Um objeto do tipo *TSuperEl* reúne as matrizes de rigidez dos elemento individuais em uma matriz de rigidez global.

Análise comparativa com o *framework* proposto

Como descrito anteriormente um elemento geométrico pode ser refinado em um conjunto de novos elementos geométricos. O conjunto de elementos refinados que representa todo um modelo geométrico ou parte deste modelo é armazenado em um modelo refinado denominado *REMModel.* Vários modelos refinados podem ser obtidos de um único modelo geométrico permitindo diferentes refinamentos ou mesmo modelos de partes distintas da estrutura como um modelo obtido a partir das vigas de um pavimento e um modelo de pórticos planos obtidos pelos pilares e vigas de extremidade da estrutura.

No caso do REMFrame um modelo refinado (classe *REMModel*) mantém um dicionário que referencia cada elemento refinado ao elemento original do modelo geométrico. Um modelo refinado pode sofrer novos refinamentos sem perder esta vinculação entre os elementos.

Os elementos de um modelo REMModel podem receber cargas (classe *Load* e derivadas) e retrições ou liberações. Estes dois elementos são representados por uma classe que implementa uma lista parametrizada armazenada em um dicionário do modelo. Existe um dicionário para restrições e outro para liberações. Os dicionários vinculam a restrição ou liberação ao elemento de aplicação através de seu identificador.

Classes do PZ que implementam a análise via elementos finitos:

A classe *TAnalysis* aciona uma seqüência apropriada de métodos para desempenhar uma análise via elementos finitos. A solução de um problema requer a criação de um objeto do tipo *TAnalysis* cuja contrutora recebe um ponteiro para uma malha computacional. Define-se uma matriz externa para ser usada como a matriz global, escolhe-se a solução a ser utilizada, a variável de pós-processamento que será retornada e o arquivo de saída, chamando-se em seguida o método *run*.

Análise comparativa com o *framework* proposto

O REMFrame possui uma classe *Façade* (item 4.2.5, sub-item b.2) que aciona um conjunto de métodos de outras classes do modelo para desempenhar um determinado caso de uso. Esta classe, semelhantemente à classe *TAnalysis*, para realizar o caso de uso *ExportData* (item 5.2.8) para transferir dados para sistemas externos, deve ser configurada com um objeto que define a formatação de dados para cada sistema externo e os arquivos utilizados (derivada de *DataManager*), um objeto que configura dados relativos a configurações para a execução deste sistema (derivada de *Domain*) e um objeto que define os métodos de acesso aos arquivos (derivada de *Filler*).

Pré-processamento e pós-processamento:

O desenvolvimento do ambiente PZ concentra-se na programação científica, utilizando pré e pós-processadores externos, ao invés de possuir bibliotecas gráficas associadas a ele.

Para realizar o pré-processamento, são definidas classes de interface que são capazes de ler arquivos produzidos por pré-processadores externos. Para visualizar os resultados são implementados filtros de saída para três pacotes gráficos e outros formatos podem ser facilmente implementados.

A classe *TGrafGrid* contém um mapa dos elementos gráficos e nós utilizados para gerar saída para os arquivos que serão lidos pelo processador externo. Uma malha gráfica é criada de uma malha computacional pela criação de um elemento gráfico de cada elemento computacional e um nó gráfico para cada grau de liberdade nodal. A classe *TGrafGrid* é uma classe virtual da qual classes específicas de saída são derivadas.

A classe *TGrafEl* é a representação lógica de um elemento uniformemente refinado. É a classe base de uma família de elementos gráficos topologicamente diferentes. Assim como os elementos computacionais, os elementos gráficos possuem um número fixo de nós gráficos que utilizam os elementos gráficos aos quais pertencem para computar sua localização ou valor.

Análise comparativa com o *framework* proposto

O REMFrame concentra-se no armazenamento, no gerenciamento e na transferência de dados de modelo para diferentes sistemas externos. A representação gráfica dos elementos fica a cargo das aplicações que devem definir as classes de visualização de cada elemento a partir da classe *View*.

A transferência de dados é implementada pelas classes derivadas das classes base *Filler*, *DataManager*. Objetos *Filler* definem os modos de acesso aos arquivos de entrada e de saída de dados, enquanto os objetos *DataManager* definem a formatação dos dados para a leitura ou gravação.

3.9.2 O Ambiente OOPAR

O OOPAR é um ambiente orientado a objetos para o desenvolvimento de sistemas para programação paralela.

A interface orientada a objetos para programação paralela apresentada desenvolve um novo paradigma que oferece uma interface para a transferência de mensagens, um modelo de dados distribuídos e um conceito de tarefa paralela. OOPAR

apresenta camadas de abstração, que permitem escrever programas paralelos não sincronizados, transmitindo estruturas de dados complexas entre os processos. Segundo o autor, as camadas de abstração são: abstração da transferência de dados, abstração do mecanismo de ponteiros distribuídos, administração de objetos distribuídos, tarefas e administração de tarefas.

A transmissão de dados entre processos é realizada através da serialização de objetos. Os dados são transformados em seqüências de bytes para transferência e depois são recompostos. Dentro do ambiente OOPAR classes serializadas são derivadas da classe base *TSaveable* que define a interface que as classes derivadas têm de implementar. Para inicializar uma recomposição de um objeto baseada em uma seqüência de bytes, um método de recomposição é associado a cada classe serializada.

Dentro de um ambiente seqüencial um objeto é identificado por seu endereço. Dentro da computação paralela com memória distribuída o mecanismo de ponteiros não pode mais ser utilizado para identificar objetos. Para solucionar o problema o OOPAR introduz o objeto de dados distribuído, o *DataManager*, que associa um identificador único a cada objeto que necessita ser compartilhado ao longo do processo. Os objetos de dados distribuídos são gerenciados pelo *DataManager* que armazena os dados em uma árvore binária para acesso rápido.

Um objeto *TTask* é capaz de acessar e transformar objetos de dados distribuídos através de seu método virtual *execute*. A classe *TTask* é derivada de *TSaveable* e os objetos *task* podem ser transmitidos de um processo para outro. Dois tipos de acesso são diferenciados: acesso de leitura e acesso de escrita. Um objeto *task* não altera um objeto de dados distribuído quando tem acesso de leitura ao mesmo. Ele somente modifica os objetos de dados distribuídos quando tem acesso de escrita aos mesmos.

Análise comparativa com o *framework* proposto

Para a transferência de dados do REMFrame, a classe base *Component* define em sua interface um método de construção de um objeto complexo (*FillIn*) e um método de gravação ou transferência de um objeto complexo (*FillOut*). Estes métodos

ao serem implementados em cada classe derivada (padrão *Strategy, GAMMA et al,* 2000) permitem que objetos complexos sejam gravados ou lidos de documentos externos, permitindo a transferência de dados entre diferentes projetos.

A transferência de dados entre sistemas externos é realizada através da interface da classe base *DataManager*. Apesar de ter recebido o mesmo nome da classe que gerencia os objetos distribuídos no OOPAR, o funcionamento destas classes não é o mesmo. Os métodos declarados na interface da classe *DataManager* do REMFrame são definidos pelas classes derivadas para implementar a importação de dados de sistemas externos, construindo ou alterando os objeto do modelo ou para a exportação de objetos do modelo para tais sistemas. Durante a importação de um objeto a classe *DataManager* tem acesso de escrita a um objeto para que o mesmo seja construído ou alterado pela inserção dos dados oriundos dos sistemas externos. Durante a exportação, o acesso ao objeto é apenas de leitura. Esta classe não armazena os objetos, mas apenas gerencia o fluxo de dados e manipula os objetos durante os processos de importação e de exportação.

3.10 BRFEM (GOPAC/UFMG *et al*, 2003) – Versão 1.1

O *Brazil Finite Elements* (BRFEM) é apresentado como um *framework* para aplicações, destinado à análise via elementos finitos de campos eletromagnéticos. Sua arquitetura foi projetada para permitir a geração de aplicações para elementos finitos nodais ou de aresta, 2D ou 3D, utilizando qualquer formulação matemática. O *framework* não faz nenhum tipo de pré ou pós-processamento e a entrada e saída de dados é realizada através de arquivos.

Nesta seção, a análise comparativa entre os *frameworks* BRFEM e REMFrame, também será apresentada ao longo da síntese devido à similaridade entre os mesmos em vários pontos apresentados.

A arquitetura do BRFEM foi construída como uma adaptação do padrão *Model View Controller* (MVC), abordado no item 4.2.4 do presente trabalho. O pacote *Model* contém o conjunto de classes que implementam a abstração dos elementos finitos, o pacote *View* agrupa as classes responsáveis pela interface externa e o pacote *Control* encapsula o fluxo de execução principal. Foi definido ainda o pacote *Error* que inclui classes de tratamento de erros.

Este *framework* utiliza-se de vários padrões de projeto para a estruturação dos relacionamentos entre as classes que compõem cada pacote. Alguns destes padrões são abordados no item 4.2.5 do presente trabalho.

Análise comparativa com o *framework* proposto

O *framework* REMFrame também utiliza uma adaptação do padrão MVC organizando suas classes em classes responsáveis por armazenar os dados do sistema estrutural, classes responsáveis por fornecer uma representação gráfica dos dados e classes responsáveis por controlar fluxos de dados e de solicitações de alteração. Por outro lado a modularização do *framework* REMFrame foi realizada segundo uma adaptação do padrão *Microkernel* (BUCHMANN eT al, 1996) descrito no item 2.4.2 desta tese. Foram definidas seis bibliotecas de vínculo dinâmico que implementam os componentes deste padrão, a saber: *Classes MVC, Filler Classes, Math Classes, Material Classes, Section Classes, Kernel.*

3.10.1 O pacote View do *framework* BRFEM

O pacote *View* contém dois tipos de classes principais: *UserInterface* e *FileInterface.* A classe *UserInterface*, derivada da classe *Observer* (padrão *Observer*, GAMMA *et al*, 2000), é utilizada para implementar diferentes interfaces de usuário. A classe *FileInterface*, é utilizada para implementar diferentes formatos de interfaces de arquivos. A FIGURA 3.17 apresenta a hierarquia destas classes.

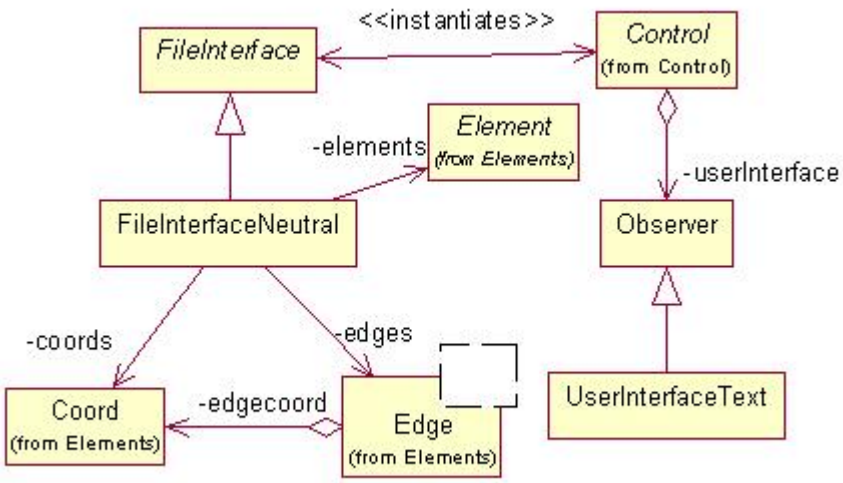

FIGURA 3.17 – Hierarquia da classe *FileInterface* do BRFEM

Análise comparativa com o *framework* proposto

Para o *framework* REMFrame, em termos de classes de visualização, foi definida uma classe base de visualização, a classe *View* que declara em sua interface os métodos *Draw* e *Erased,* que permitem a definição das visualizações dos elementos de dados pelas aplicações.

Para permitir diferentes formas de entrada de dados, foi definida a classe base *Filler.* As classes concretas derivadas de *Filler*, podem especializar os métodos leitura e de gravação de dados para diferentes ambientes, como foi realizado para a tranferência de dados via plataforma gráfica nas aplicações desenvolvidas.

No REMFrame a interface de arquivos é implementada pela classe *FileFiller* derivada de classe *Filler*, juntamente com classes derivadas de *DataManager*. A definição dos formatos de arquivos gerados são então realizados pelas aplicações através das classes concretas derivadas de *DataManager*.

3.10.2 Pacote *Control* do BRFEM

No pacote *Control* encontramos a hierarquia da classe *Control* que encapsula o fluxo principal de execução. Esta classe foi estruturada segundo o padrão *Factory method* (GAMMA et al, 2000) e seu método *specialize* retorna um objeto *Control* específico para diferentes problemas: linear, não linear, etc. A hierarquia da classe *Control* é apresentada na FIGURA 3.18.

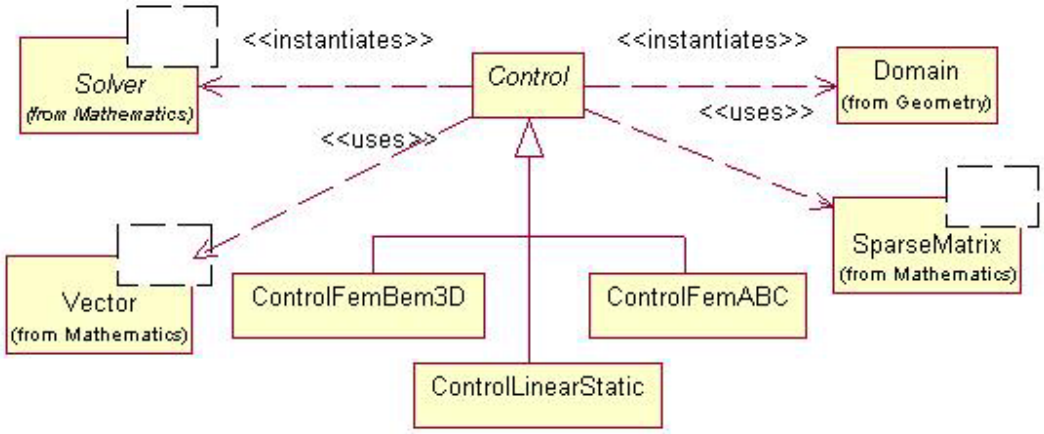

FIGURA 3.18 - Hierarquia da classe *Control* do BRFEM

Análise comparativa com o *framework* proposto

No framework REMFrame a classe de controle geral do sistema, que implementa os casos de uso definidos, é a classe *ProjectController*. De maneira semelhante à classe *Control* do BRFEM, a classe *ProjectController* deve ser configurada com objetos de outras classes para implementar a funcionalidade geral da aplicação gerada com o uso do *framework* REMFrame. No REMFrame um objeto do tipo *ProjectController* pode receber um objeto para definição dos métodos de leitura e gravação de dados (*Filler*), objetos do tipo *Factory* para criação dos objetos de visualização (*View*), de construção (*Builder*), de formatação de dados (*DataManager)* e de formatação de ambiente (*Domain*), específicos da aplicação desenvolvida com o uso deste *framework*.

3.10.3 Pacote Model do BRFEM

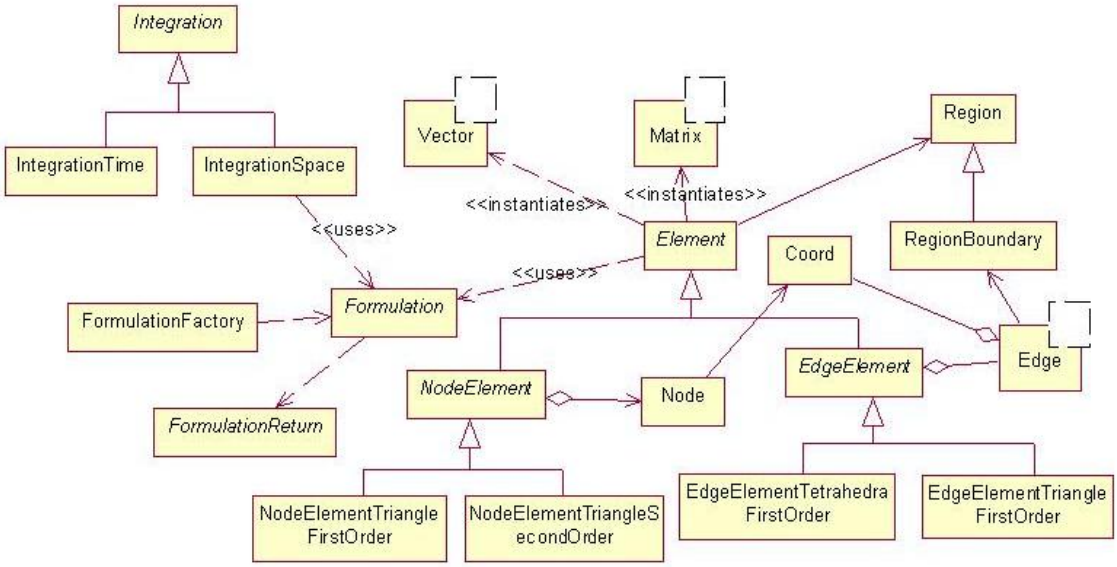

FIGURA 3.19 – Classes do pacote *Model* do BRFEM

As classes do pacote *Model* do BRFEM foram organizadas em cinco pacotes, descritos a seguir.

O pacote *Element*:

Esse pacote contém a hierarquia das classes *Element* e *Formulation*. A desvinculação entre a representação do elemento finito e da formulação matemática utilizada permite a alteração desta última sem afetar a implementação dos elementos e vice versa.

A classe *Element* é uma classe abstrata, base para a definição de todos os elementos do sistema. Cada elemento é responsável pela determinação de sua matriz local ou por retornar as funções de forma em um dado ponto. Um elemento está associado a um objeto do tipo *Region* que contém as definições de materiais, de fontes e de formulações.

Análise comparativa com o *framework* proposto

No *framework* REMFrame foi também definida a classe *Element* cuja hierarquia possui uma similaridade à hierarquia da classe de mesmo nome definida no BRFEM. No REMFrame esta classe é derivada de Subject representando, assim uma classe de modelo e está vinculada a uma classe contêiner (*Model*) a qual contém as informações sobre todos os outros objetos que compõem uma representação da estrutura. Esta classe está inserida no pacote *Mesh Classes,* do componente *Kernel* do REMFrame.

O pacote *Mathematics*:

Este pacote reúne as classes que representam entidades matemáticas (FIGURA 3.20): matrizes vetores, etc. Estas classes utilizam classes da biblioteca padrão da linguagem C++, *vector*, *map* e *valarray,* para obter armazenamento e acesso a dados de forma eficiente. A classe *Assembler* é uma classe abstrata que define a montagem do sistema de equações. As classes da hierarquia *Integration* são responsáveis por implementar as integrações no sistema. As classes *Solver* são responsáveis por solucionar os sistemas de equações lineares ou não lineares obtidos.

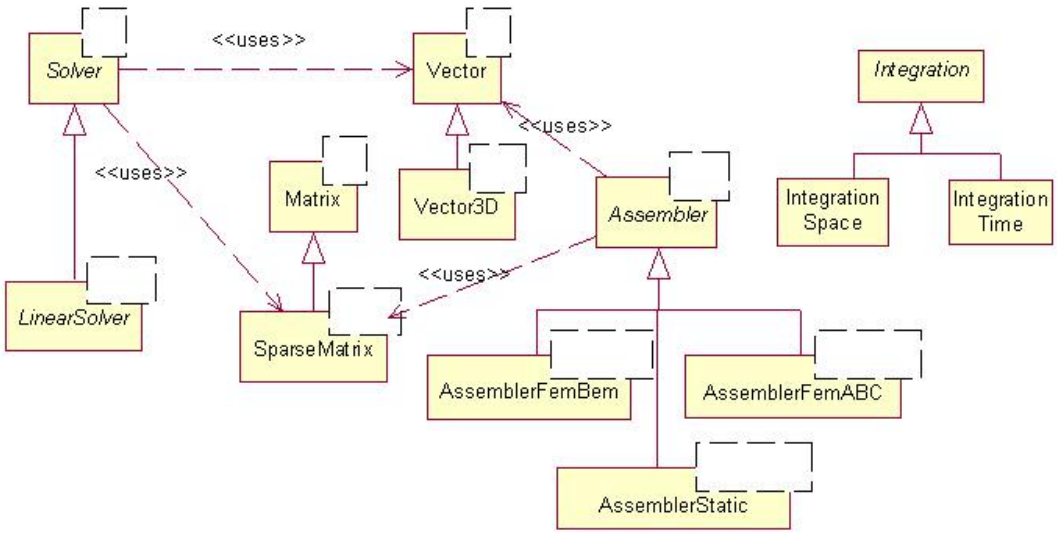

FIGURA 3.20 – Classes do pacote *Mathematics* do BRFEM

Análise comparativa com o *framework* proposto

O *framework* REMFrame também define um componente (*Math Classes*) que reúne as classes que representam entidades matemáticas e geométricas, utilizando as classes *map, vector* e *list* da biblioteca padrão da linguagem C++ para armazenamento de dados. Estas classes permitem ao *framework* manipular e gerenciar seus dados geométricos, sem depender da plataforma gráfica para tal tarefa. As classes deste componente são: *Point*, *Line*, *Axis*, *CoordinateSystem*, *Plane*, *Matrix*, *Interpolation e Values.* A classe *Values* implementa um vetor de dados parametrizado.

O pacote *Physics*:

Este pacote contém classes que modelam as características físicas do problema. As classes deste pacote foram implementadas para possibilitar a criação de materiais com diferentes combinações de propriedades isotrópicas, anisotrópicas, lineares ou não lineares. A classe *Material* possui um mapa que associa a cada nome de propriedade um vetor que armazena as características de uma propriedade em diferentes direções. A classe *CurveModel* implementa o tratamento linear ou não linear das propriedades. A classe *Source* é uma classe abstrata que representa as fontes do problema eletromagnético: correntes, cargas elétricas, etc. As classes do pacote *Physics* são apresentadas na FIGURA 3.21.

Análise comparativa com o *framework* proposto

A classe *Material* do *framework* REMFrame possui uma estrutura semelhante à classe *Material* do BRFEM, contando com a classe *PropertyMap* para a definição das propriedades físicas. Estas classes são encontradas no componente *MaterialClasses*.

Para implementar a interpolação de valores segundo uma função polinomial foi declarada a classe *Interpolation* inserida no componente *MathClasses* do REMFrame. Esta classe pode ser utilizada para representar distribuições não lineares de valores como, por exemplo, a distribuição de cargas.A proposição desta classe foi baseada na classe *CurveModel* do BRFEM.

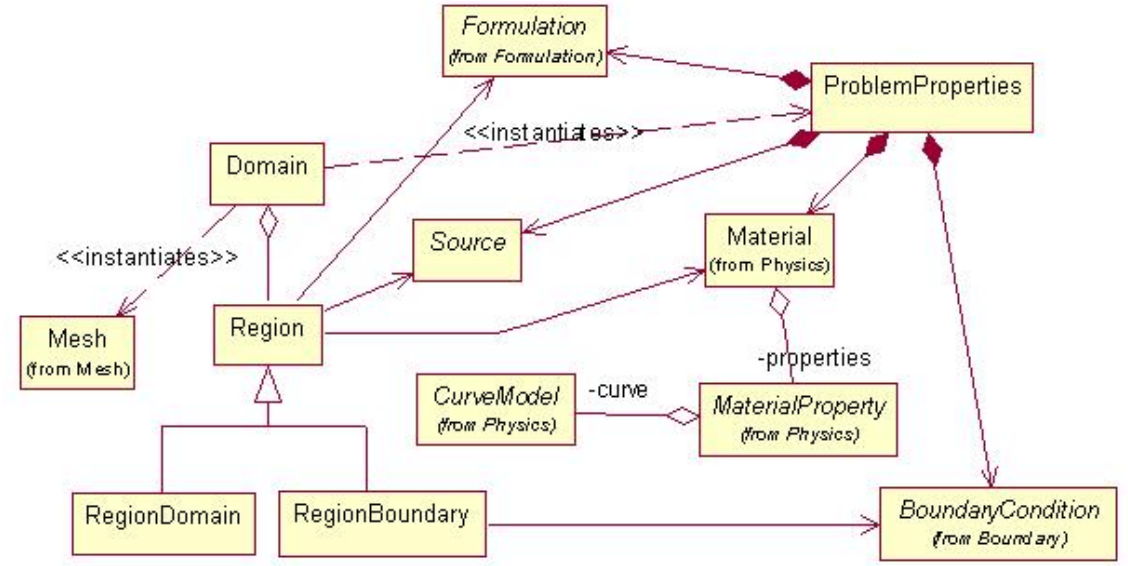

FIGURA 3.21 – Classes dos pacotes *Physics* e *Geometry* do BRFEM

O pacote *Mesh*:

A classe *Mesh* é uma classe de banco de dados que armazena informações sobre todos os elementos do modelo: elementos, nós, arestas e suas conectividades (FIGURA 3.22).

Análise comparativa com o *framework* proposto

No REMFrame a classe que reúne os elementos do modelo, neste caso as barras, os nós, as conecções, os carregamentos, as restrições, as liberações, e os valores prescritos também é uma classe de banco de dados. A classe base desta hierarquia é a classe *Model.* O *Model* armazena também uma listagem dos identificadores de materiais e de seções aplicados aos seus elementos.

O pacote *Geometry*:

Este pacote contém as classes que tratam a geometria do problema: *Region* e *Domain* (FIGURA 3.21). A classe *Domain* armazena a definição do domínio do problema: as regiões existentes, a malha (*Mesh*) e as propriedades do problema (*ProblemProperties*). A classe *Region* contém as definições de materiais, fontes e formulações.

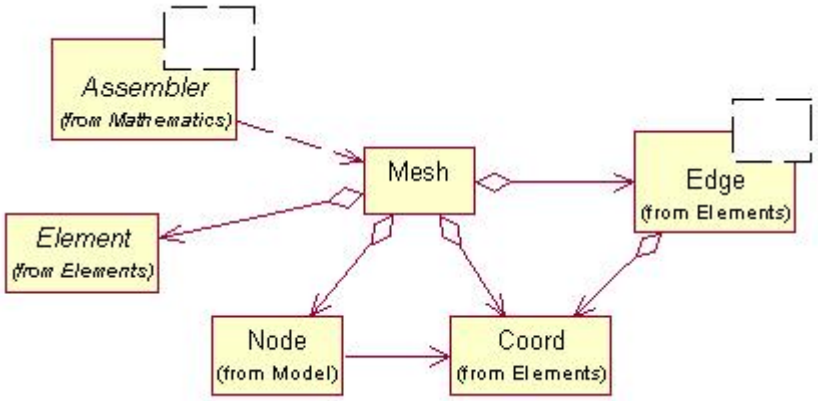

FIGURA 3.22 – Estrutura e relacionamento da classe *Mesh* do BRFEM

Análise comparativa com o *framework* proposto

No REMFrame os dicionários de materiais e de seções são armazenados em outra classe de banco de dados, a classe *Project* que possui também um dicionário dos modelos existentes no projeto: modelo geométrico e modelos refinados, um dicionário dos modelos de interpolação definidos (classe *Interpolation*), e um dicionário das definições de domínio dos sistemas externos utilizados (classe *Domain*).

As classes de banco de dados do REMFrame, entre elas as classes *Model* e *Project*, foram organizadas no componente *DataBaseClasses*.

3.10.4 *Hot spots*

Os *hot spots*[13] do *framework* são classes abstratas, localizadas no núcleo do sistema e que permitem a criação de novos programas de elementos finitos pela instanciação de classes concretas a partir destas classes. A FIGURA 3.23 apresenta os *hot spots* do *framework*.

São citadas três implementações de sistemas a partir deste *framework.* Na primeira implementação (um sistema para resolver sistemas 2D utilizando o vetor potencial magnético) foram instanciadas classes concretas para os hot spots *Control*, *UserInterface*, *FileInterface*, *Element*, *Formulation*, *Assembler*, *Integration* e *MaterialProperty.* Na segunda implementação (um sistema para resolver problemas bidimensionais usando elementos finitos de segunda ordem) foram utilizados todas as implementações citadas anteriormente, tendo sido acrescentada a classe *NodalTriangleSecondOrder* derivada do hot spot *Element*. A terceira implementação (um sistema para problemas de arestas tridimensionais dispersas via método hibrido de elementos finitos-elementos de contorno) foram instanciadas novas classes concretas para representar os elementos de borda (*hot spot Element*), para formulação do método hibrido (*hot spot Formulation*), para montagem da matriz do sistema (*hot spot Assembler*) e para definir o fluxo geral do sistema (*hot spot Control*). Estes três sistemas instanciados de uma maneira fácil a partir da utilização do *framework* mostraram as vantagens deste processo de desenvolvimento, onde novos sistemas podem ser desenvolvidos pela instanciação dos *hot spots* definidos.

---

[13] Pontos de flexibilização do sistema que devem ser customizados para cada aplicação específica.(MARKIEWICS e LUCENA, 2001)

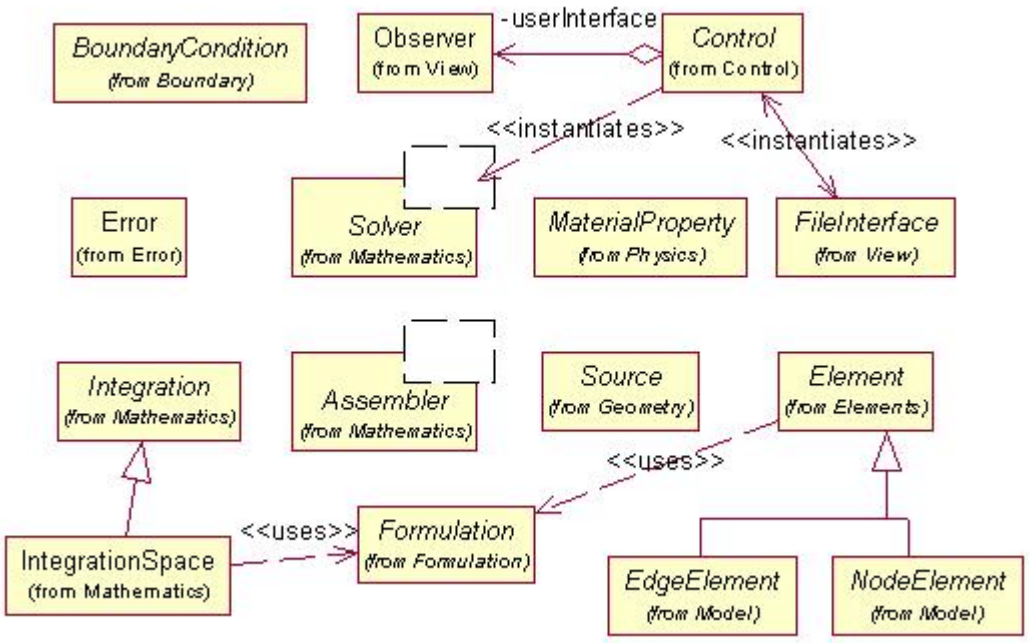

FIGURA 3.23 – *Hot spots* do *framework* BRFEM

Análise comparativa com o *framework* proposto

Os *hot spots* definidos para o REMFrame visam permitir a flexibilização de representações de criação de objetos e principalmente permitir a integração entre sistemas através da flexibilização da transferência de dados. Os *hot spots* do REMFrame e as aplicações desenvolvidas são apresentados no item 5.2 e capítulo 6, respectivamente.

3.11 INSANE (ALMEIDA, 2005, GONÇALVES, 2004, GONÇALVES e PITANGUEIRA, 2004, FONSECA, 2006, MOREIRA, 2006)

O *Interactive Structural Analysis Environment* (INSANE) é um projeto de *software* livre desenvolvido na linguagem de programação Java, organizado em três segmentos: pré-processador, processador e pós-processador. O sistema, estruturado

segundo o padrão de arquitetura *Model View Controller* (MVC), abordado no item 4.2.4.do presente trabalho, permite a análise estrutural via modelos de elementos discretos. GONÇALVES (2004) apresenta uma estrutura de classes para o gerador de malhas do sistema e ALMEIDA (2005) apresenta uma análise orientada a objetos para a formulação paramétrica do MEF, implementada no núcleo numérico do INSANE. FONSECA (2006) apresenta uma ampliação do processador com a inclusão da solução de modelos estruturais reticulados fisicamente não lineares, através do MEF. MOREIRA (2006) apresenta uma aplicação gráfica interativa que permite a visualização de informações referentes à solução de modelos do MEF através do núcleo numérico do sistema.

A revisão destes trabalhos focou a estrutura de classes utilizada para a representação da geometria do sistema e para a análise numérica, procurando similaridades e diferenças em relação aos sistemas citados anteriormente. Também o funcionamento geral dos sistemas de pré e pós-processamento foi estudado sob o ponto de vista do usuário, avaliando a forma de entrada de dados, a ordem de execução, as saídas gráficas e as formas de persistência.

A estrutura de classes do projeto é bastante extensa, principalmente para os sistemas gráficos, que representam plataformas independentes, desenvolvidas pelo grupo. Apresenta-se, então, de forma resumida, as classes do núcleo numérico do sistema consideradas relevantes para o problema tratado na presente tese.

A interface *Assembler* é responsável por montar o sistema matricial de segunda ordem utilizado para representar os problemas discretos[14]. A interface *Solution* define os

---

[14] MARTHA, L. F. ; MENEZES, I. F. M. ; LAGES, E. N. ; PARENTE JUNIOR, E. ; PITANGUEIRA, R. L. S. . An OOP Class Organization for Materially Nonlinear Finite Element Analysis. In: Join Conference

métodos para solucionar as equações dos possíveis modelos. A interface *Persistence* trata da persistência de dados tratando dados de entrada e gravando dados de saída em objetos de persistência.

A interface *Model* representa o modelo discreto a ser analisado. *Model* é implementada pela classe *FemModel* (FIGURA 3.25). Esta classe possui como atributos listas de nós, de elementos, de funções de forma representadas pela classe *Shape*, de ordem de integração (*IntegrationOrder*), de combinações de carregamentos, de modelos de análise, de materiais, de modelos constitutivos e de degenerações.

A classe *SteadyState*, implementação da interface *Solution* é responsável pela solução de problemas lineares e a classe *EquilibriumPath* pela solução de problemas não lineares através de um processo incremental-iterativo, representado pela classe *Step*. A interface *InteractiveStrategy* representa métodos de controle. A classe *LinearEquationSystem* é responsável pela solução de um sistema de equações algébricas lineares. A FIGURA 3.24 apresenta a estrutura do núcleo da sistema INSANE.

As classes *Element, Node* e *Material* têm definições semelhantes às classes de mesmo nome implementadas nos sistemas de análise apresentados anteriormente. A classe *IntegrationOrder* representa uma ordem de integração numérica. A classe *Shape* define as funções de forma com suas derivadas para os elementos finitos. A classe *MaterialPoint* representa um ponto no meio material. A interface *ConstitutiveModel* é responsável por montar as matrizes constitutivas e calcular as tensões das degenerações e dos pontos materiais, através de informações de suas geometrias, materiais e modelos

---

of Italian Group of Computational Mechanics and Ibero-Latin American Association of Computational Methods in Engineering, 1996, Pádua, Itália. Proceedings of Join Conference of Italian Group of Computational Mechanics and Ibero-Latin American Association of Computational Methods in Engineering, 1996. v. 1. p. 229-232.

de análise. A classe *Degeneration* representa a degeneração da geometria do elemento. A classe *AnalysisModel* fornece informações sobre o modelo de análise para elementos finitos, pontos materiais e representações. A interface *ProblemDriver* informa as contribuições de cada elemento na equação do modelo sendo especializada para cada tipo de problema modelado pelas suas implementações. A classe *LoadCombination* representa as possibilidades de combinação entre os carregamentos.

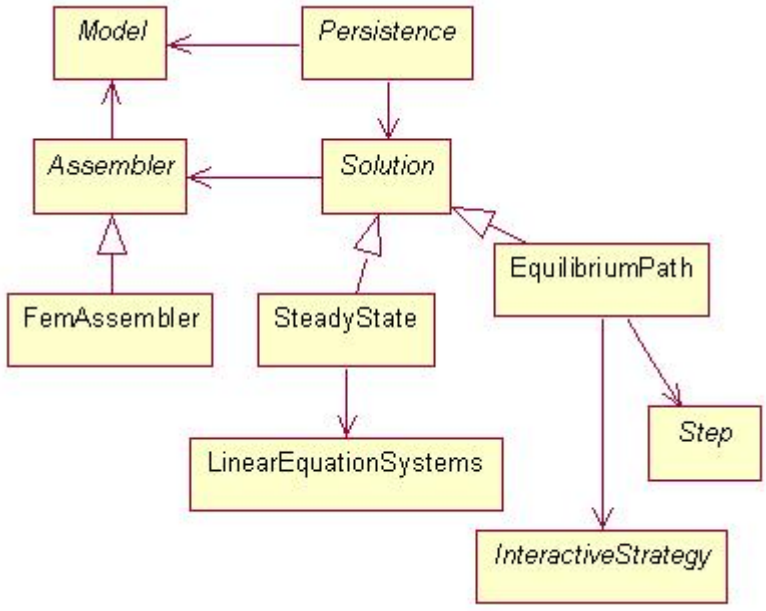

FIGURA 3.24 – Estrutura do núcleo do sistema INSANE

Análise comparativa com o *framework* proposto

Observa-se, através deste projeto que, independente da linguagem adotada, a orientação a objetos aplicada ao MEF determina um conjunto de classes que, apesar de algumas variações identificadas em cada sistema, apresentam em linhas gerais a mesma filosofia, aplicada a contextos diferentes. Também aqui identifica-se classes que representam elementos essenciais do MEF: *Element, Node* e *Material*. Objetos destas classes são armazenados na classe *FEMModel*. Esta classe é derivasda da classe *Model* que representa os modelos para diferentes tipos de análise. O paralelo entre as classes semelhantes a estas e as do REMFrame já foi apresentado na análise comparativa dos sistemas anteriores.

Enquanto a persistência de dados no INSANE é obtida através da classe *Persistence,* a persistência de dados no REMFrame é realizada através das classes derivadas da classe *Filler* e por cada classe da dados derivada de *Component* (métodos *FillIn* e *FillOut*).

De forma semelhante ao realizado pelas classes *LoadCase* e *LoadCombination* do INSANE, as classes *LoadCase* e *LoadMethod* do REMFrame representam um conjunto de cargas e o método de combinação das mesmas.

As demais classes do projeto são direcionadas para a solução do problema, quer seja de análise quer seja do sistema gráfico, e assim não possuem uma correlação imediata com a estrutura implementada pelo REMFrame.

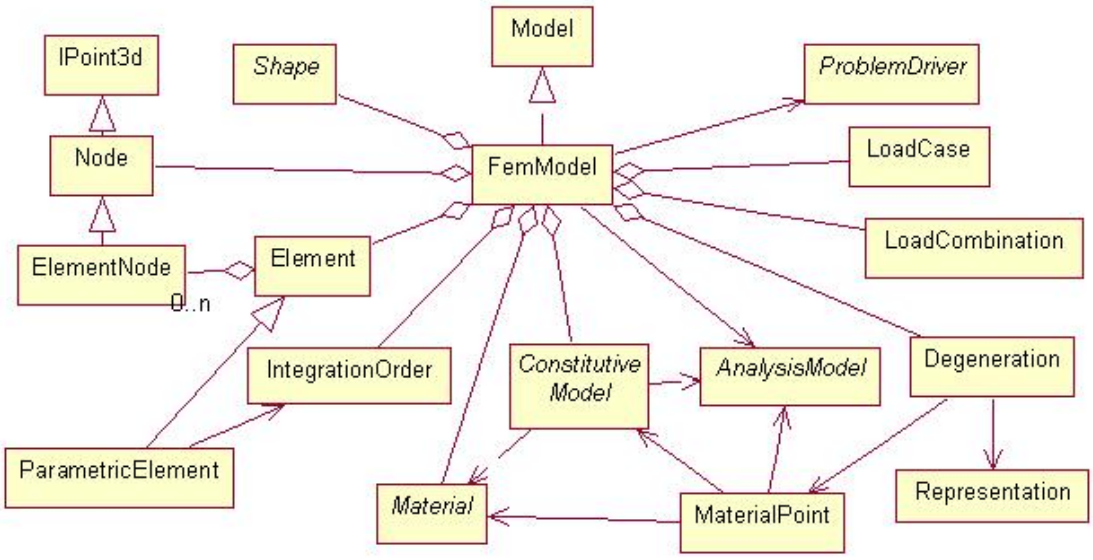

FIGURA 3.25 - Estrutura da classe *FemModel*

3.12 Modelador Estrutural CAD e *WinCollapse* (SOARES, 2006)

SOARES (2006) associou as tecnologias CAD/CAE permitindo a automação da etapa de modelagem da estrutura por um sistema CAD denominado Modelador Estrutural CAD, integrado a um aplicativo CAE para análise limite de pórticos planos via elementos finitos (FIGURA 3.26). O sistema CAD foi desenvolvido a partir do Modelador 3D reformulado, apresentado no item 3.6.

A presente tese e o trabalha de SOARES foram desenvolvidos segundo filosofias bem diferentres, ou seja: um sistema CAD desenvolvido para uma determinada plataforma CAD integrado a um determinado sistema CAE e um *framework* para integração de diferentes sistemas de um processo. Porém como ocorreram paralelamente, em um mesmo período temporal, e como havia inicialmente a intenção de se testar a aplicação do *framework* REMFrame no desenvolvimento de um sistema que utilizasse o núcleo numérico do *WinCollapse* como seu sistema de análise, muitas decisões de projeto de ambas as pesquisas são semelhantes.

3.12.1 O modelador Estrutural CAD

A estrutura final de classes do Modelador Estrutural CAD é apresentada na FIGURA 3.27. A estrutura original do modelador apresentada na FIGURA 3.13 sofreu algumas alterações, mas sem perder sua essência. Foram implementadas classes para o armazenamento de informações sobre camadas de desenho, sobre os parâmetros de desenho, sobre propriedades geométricas de seções transversais e sobre os materiais. As principais alterações foram realizadas nos métodos implementados para inserção de elementos, controle de consistência entre elementos, inclusão de nós intermediários e representação da estrutura deformada a partir dos dados resultantes da análise limite.

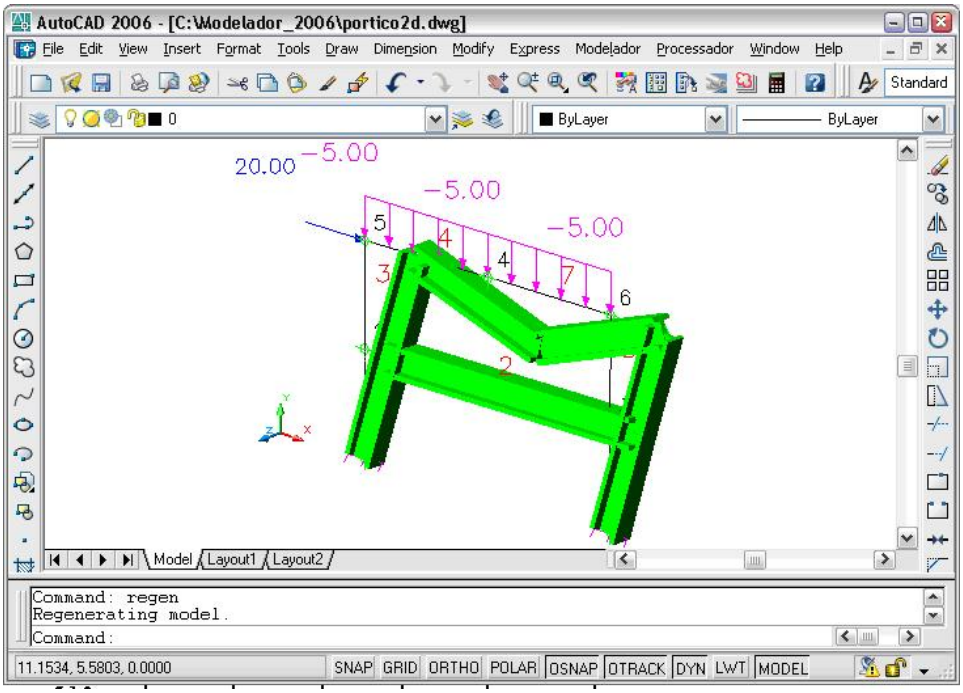

FIGURA 3.26 - Modelo sólido de pórtico plano deformado

(figura extraída de SOARES, 2006)

A classe *Initialise* contém os procedimentos que são acionados durante o carregamento da aplicação modelador ou durante a abertura ou criação de um novo arquivo de desenho no AutoCAD, uma vez que o sistema permite que diferentes arquivos sejam trabalhados em uma mesma seção da plataforma gráfica.

A classe *Layers* trata da persistência de camadas de desenho que são utilizadas pelo modelador para inserir as entidades personalizadas.

A classe *Material* contém as propriedades físicas do material a ser associado aos objetos do tipo *Bar*. A classe *Isotropic*, derivada de *Material*, foi implementada para representar materiais isotrópicos.

A classe *Section* representa as seções transversais, calculando algumas propriedades geométricas destas seções. Diferentes formas de seções transversais são representadas pelas classes derivadas instanciadas.

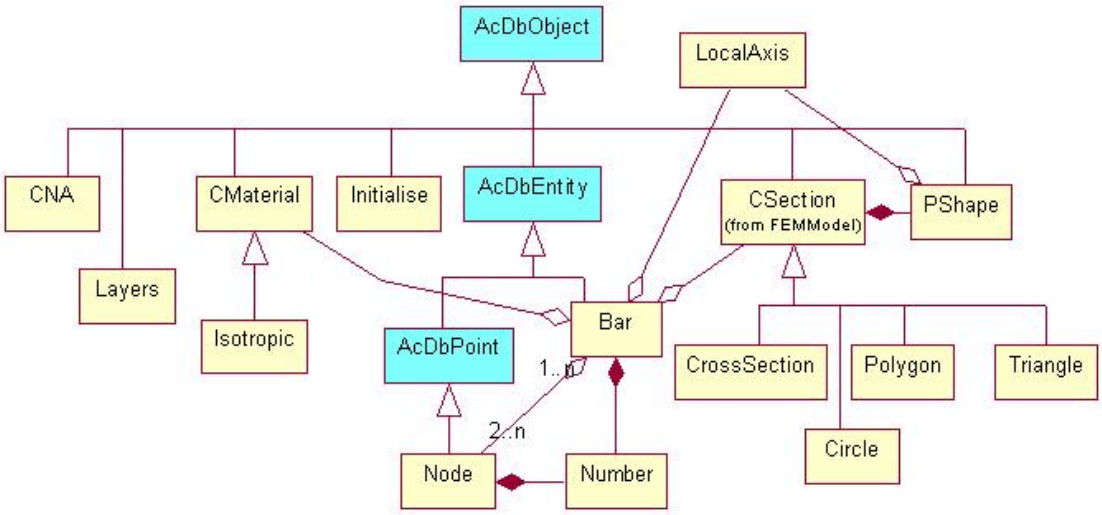

FIGURA 3.27 – Classes do Modelador Estrutural CAD

Neste sistema os carregamentos são representados por atributos das classes *Bar* e *Node*. As restrições de apoio e a posição deslocada são abstraídos por atributos da classe *Node*. A classe *Node* é derivada de *AcDbPoint*, classe nativa que representa a entidade gráfica ponto. Objetos das classes *Bar* e *Node* gerenciam sua representação gráfica composta por representações de cada um de seus atributos.

Com relação à interface de usuário, esta recebeu vários comandos acionados por menus de tela ou barras de ferramentas. Cada comando recebeu uma caixa de diálogo. Estas últimas foram desenvolvidas através da API MFC (MICROSOFT, 2005) e são utilizadas para editar as propriedades dos nós, das barras, das seções transversais e dos materiais e configurar os modos de exibição dos componentes gráficos das entidades, como: textos representativos da numeração, geometria unifilar ou sólida, representação de apoios, carregamento, etc.

Análise comparativa com o *framework* proposto

No REMFrame todas as classes de representação gráfica são derivadas da classe *View* e devem ser implementadas pelas aplicações. A aplicação *GeometricModeler*

(item 6.1 desta tese) faz a implementação de um modelador geométrico semelhante ao Modelador Gáfico 3D.

Para a implementação das representações dos objetos *Bar*, *Node*, *Load*, *Section*, e de atributos da classe Model como restrições e liberações, deslocamentos prescritos, foram definidas várias classes derivadas da classe *View* (item 6.1.1). Foram também derivadas classes de visualização para representar as deformadas da estrutura e os esforços resultantes aplicados em cada elemento refinado, obtidos como dados de saída dos sistems de análise integrados pelo *framework*. Para cada uma das classes da hierarquia de *View* foi criada uma classe derivadas da biblioteca gráfica da plataforma para a representação dos elementos Cada representação gráfica é então resultado da conbinação destas duas hierarquias (*View* e *AcDb*) através do padrão *Builder* (GAMMA et al, 2000).

A implementação de comandos, de interface de usuário também é apresentada no item 6.1.1. Os comandos são definidos através da classe *ARXCommand* derivada da classe *Command* do núcleo do *framework*. O controle geral do sistema é realizado pela classe *ARXProjectController* derivada da classe *ProjectController* também do *framework.*

3.12.2 O sistema de análise

O sistema CAE desenvolvido, denominado *WinCollapse*, foi estruturado em dois módulos principais: o módulo *FEMModel,* que implementa o modelo de elementos finitos utilizado pelo módulo de análise, e o módulo *Collapse,* que implementa a formulação para análise limite de pórticos planos.

O módulo *FEMModel* (FIGURA 3.28) é formado pelas classes *CModel*, *CNode*, *CElement*, *CSection* e *CMaterial*. Estas classes representam as entidades básicas do modelo discreto de elementos finitos para análise limite de pórticos planos, e receberam o prefixo “C” para diferenciá-las das classes do Modelador Estrutural CAD. A classe *CNode* faz a abstração dos nós da estrutura, e seus atributos são: as coordenadas nodais, o carregamento aplicado, os deslocamentos nodais, os graus de liberdade e as restrições

de apoio aplicadas. A abstração dos elementos finitos lineares é realizada pela classe *CElement*. Esta classe possui, além de atributos para representar o carregamento e os graus de liberdade ativos, uma coleção de objetos ou instâncias da classe *CNode*, um objeto *Material* e um objeto *CSection*. A classe *CSection* representa as seções transversais aplicadas ao objeto *CElement*. Seus atributos representam as propriedades geométricas dessas seções. A classe *CMaterial* representa os materiais aplicados aos elementos lineares. Cada objeto do tipo *CMaterial* armazena dados sobre as propriedades físicas a ele relacionadas. A classe *CModel* foi desenhada para representar o modelo estrutural, armazenando todos os elementos que compõem uma determinada estrutura através de listas encadeadas de objetos das classes *CElement*, *CNode*, *CMaterial* e *CSection*.

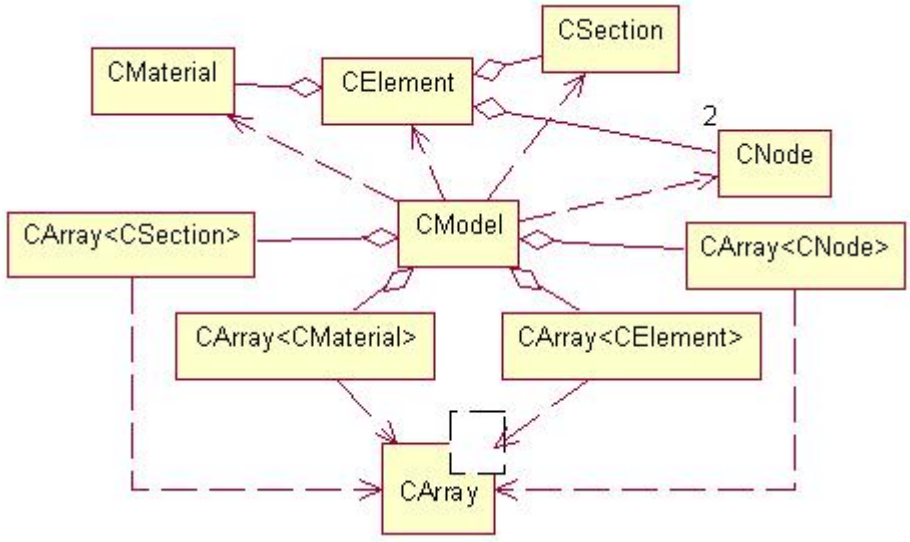

FIGURA 3.28 – Classes do Modelador 2006: componente *FEModel*

O módulo *Collapse* (FIGURA 3.29) apresenta uma estrutura de classes que abstrai os componentes necessários para um sistema genérico de análise estrutural. As classes apresentadas, com exceção de *Driver* e *Model*, são classes abstratas. Para cada uma destas classes foram implementadas uma ou mais classes concretas para realizar a análise do sistema em abordagem cinemática via programação linear, utilizando o

algoritmo desenvolvido por FRANCO e PONTER[15] *apud* SOARES (2006), e através do método da combinação de mecanismos.

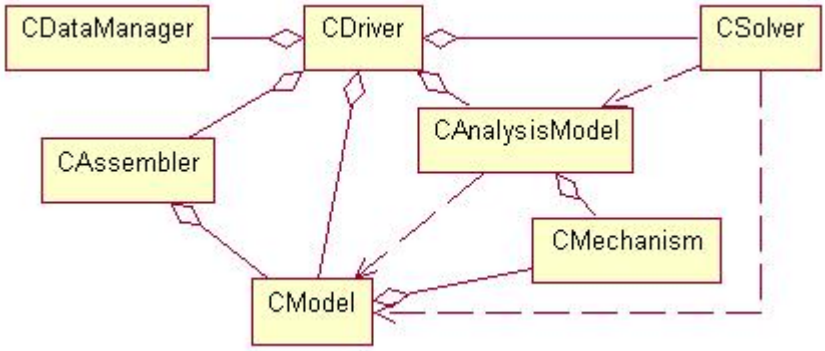

FIGURA 3.29 – Classes do componente *Collapse*

O fluxo dos procedimentos interativos entre as classes deste componente, para realizar o processamento da análise estrutural, foi apresentado por SOARES na forma de um diagrama de seqüência[16].(FIGURA 3.30)

Para executar a análise limite de pórticos planos, desenvolveu-se a classe *RigidBeam,* derivada da classe *CElement,* que representa o elemento finito de viga

---

[15] FRANCO J.R.Q. and PONTER A.R.S, "A general approximate technique for the finite element shakedown and limit analysis of axisymmetrical shells – Part I – Theory and fundamental relations", *International Journal for Numerical Methods in Engineering*, Vol. 40, pp. 3495-3513, (1997).
FRANCO J.R.Q. and PONTER A.R.S, "A general approximate technique for the finite element shakedown and limit analysis of axisymmetrical shells – Part II – Numerical applications", *International Journal for Numerical Methods in Engineering*, Vol. 40, pp. 3515-3536, (1997)

[16] O diagrama de seqüência é um diagrama da UML que mostra as interações entre elementos em uma sequancia temporal (QUATRANI, 1998).

rígida. A classe *CAssembler* realiza a montagem das matrizes, dos vetores e dos objetos associados ao problema estrutural, acessando o modelo estrutural através de um objeto do tipo *CModel*, e o modelo de análise através do objeto do tipo *CAnalysisModel*. A classe *Solver* abstrai o processo de solução do problema estrutural. O gerenciamento dos dados, envolvendo a leitura de dados de entrada e a persistência dos resultados é realizada pela classe *CDataManager*. O gerenciamento geral da execução do sistema e da interação entre as classes é realizado pela classe *CDriver*.

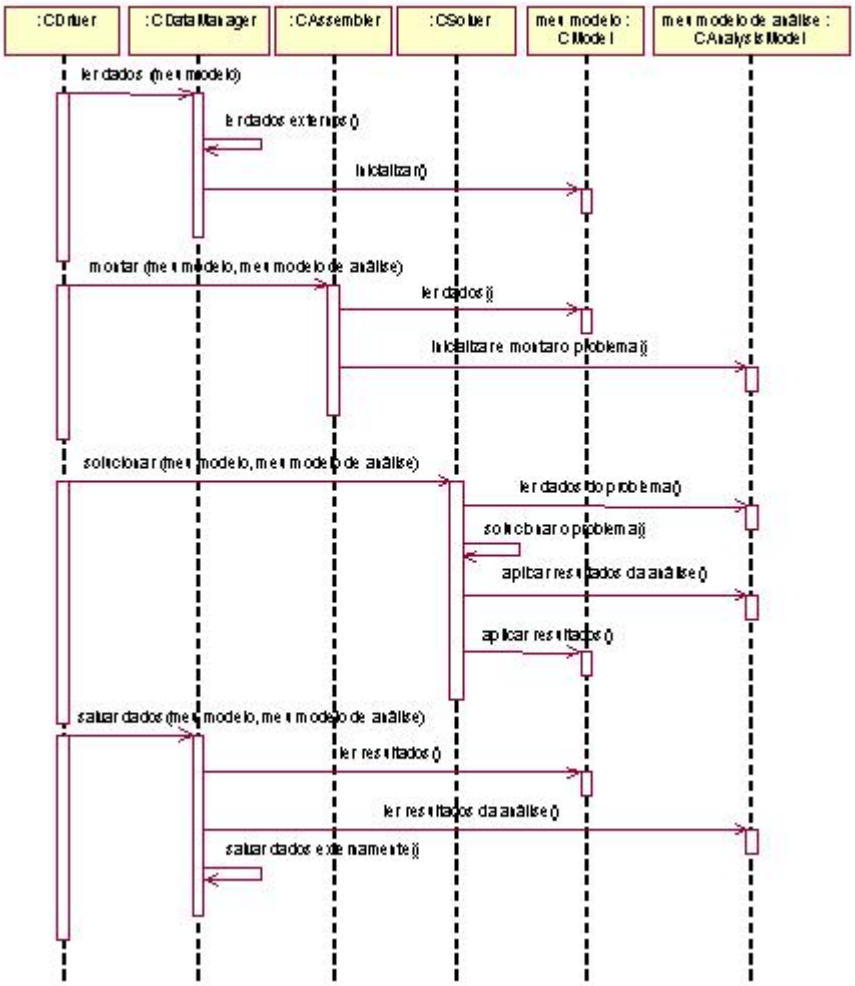

FIGURA 3.30 – Diagrama de seqüência: Funcionalidade geral do sistema *WinCollapse* (figura extraída de SOARES, 2006)

Análise comparativa com o *framework* proposto

Assim como o Modelador Gráfico 3D o REMFrame apresenta uma classe *Section* que realiza a abstração das seções transversais aplicadas ao objeto *Bar*. Para a definição das propriedades geométricas das seções ambos trabalhos utilizaram o PROPFIG, um sistema que realiza o cálculo de diversas propriedades geométricas de seções transversais simples e compostas. Inicialmente implementado na linguagem Pascal ele foi re-implementado segundo a POO para ambos os sistemas. Soares implementou o tratamento das propriedades geométricas em uma hierarquia de classes definida a partir da classe base *Section*, com representação apenas das seções simples. Na presente tese separou-se representação geométrica, realizada pela hierarquia da classe *Shape*, do cálculo das propriedades geométricas implementada na hierarquia da classe *GeomPropHandler*. A classe *Section* é capaz de representar seções compostas formadas por um conjunto de objetos do tipo *Shape*, retornando as propriedades geométricas da seção composta através do somatório das propriedades de cada objeto *Shape* que possui uma referência para um objeto do tipo *GeomPropHandler.*Nesta organização foram utilizados os padrões *Bridge* e *Strategy* (GAMMA et al, 2000).

As classes do módulo *FEMModel* apresentam uma forte semelhança com as classes implementadas pelo REMFrame, porém a estrutura final de relacionamentos apresenta algumas diferenças. No REMFrame a classe *Model* não armazena os dicionários de materiais e de seções, que ficam a cargo da classe *Project*. Um objeto do tipo *Bar* não possui referências diretas para nós, material e seções, armazenando apenas os identificadores dos elementos a ele aplicados.

Em relação ao módulo *Collapse*, o REMFrame apresenta a classe *DataManager* que realiza o gerenciamento dos dados, semelhantemente ao realizado pela classe *CDataManager*. Podemos traçar um paralelo entre as funcionalidades gerais da classe *CDriver* do *Collapse* e a classe *ProjectController* do REMFrame, uma vez que o gerenciamento geral da execução fica a cargo desta última, que realiza o repasse de solicitações das aplicações para as diversas classes de dados do *framework*.

3.13 Comparativo geral entre os trabalhos acima descritos

A análise conjunta destes sistemas, ambientes ou projetos mostra que, para ser capaz de representar os dados de uma estrutura reticular de forma genérica, mas consistente com os sistemas de análise via MEF, um sistema deve prever classes específicas para algums elementos essenciais tais como: elementos, nós, materiais, carregamentos, propriedades geométricas. Estes dados são geralmente organizados em classes contêineres como as classes: *Domain* (FEM_OBJ), *TGeoGrid* e *TCompGrid* (PZ), *Mesh* (BRFEM), *Model* (INSANE) e *CModel* (*WinCollapse*). Em alguns casos mais de uma classe contêiner podem ser necessárias, sendo geralmente derivadas de uma classe base: classe *Model* do INSANE e *CModel* do *WinCollapse*. O conjunto de objetos destas classes pode ser reunido em uma classe que realiza a configuração do domínio de atuação: *Domain* (BRFEM).

A persistência de dados e a transferência de dados é outro fator comum a todos os sistemas sendo desempenhada pelas classes *FileReader* (FEM_OBJ), *TPZStream* (PZ), *TSaveable* (OOPAR), *FileInterface* (BRFEM), *Persistense* (INSANE), *CDataManager* (*WinCollapse*).

4

METODOLOGIA

Neste capítulo é apresentada a metodologia empregada no desenvolvimento do trabalho sob dois aspectos: o referencial teórico e os fundamentos metodológicos utilizados. No referencial teórico são abordados os conceitos e as teorias principais que serviram de base para o desenvolvimento da pesquisa. Nos fundamentos metodológicos são apresentadas as técnicas e os processos adotados, assim como os instrumentos utilizados.

4.1 *Framework*

Um "*application framework"* é uma coleção integrada de componentes e de relacionamentos aplicáveis a *software* orientados a objeto. Esta coleção de componentes oferece uma arquitetura e funcionalidades predefinidas que podem ser utilizadas na construção de *software* dentro de uma área de aplicação.

O emprego de métodos numéricos e computacionais na busca por soluções mais rápidas, confiáveis e econômicas para problemas em todas as áreas do conhecimento tem tomado proporções cada vez maiores nas últimas décadas. Conjuntos cada vez mais extensos de dados e de informações têm de ser tratados, gerando sistemas refinados, cujo controle e manutenção exigem uma atenção especial, e cuja implementação apresenta sempre custos elevados.

É notório o avanço no desenvolvimento de tecnologias de *hardware*, em linguagens de programação, em padrões, em bibliotecas de classes, em componentes, etc. Por outro lado, é bastante comum, no desenvolvimento de aplicativos, que tempo e esforços dedicados na busca de soluções para problemas de baixo nível não sejam devidamente reaproveitados em trabalhos semelhantes. O custo de sistemas computacionais está intimamente relacionado aos esforços de desenvolvimento. Segundo FAYAD e SCHMIDT (1997), a maior parte dos custos e dos esforços de desenvolvimento origina-se do contínuo redescobrimento, ou reinvenção, de conceitos básicos e de componentes pela indústria de *software*. Assim, uma maneira de reduzir os custos e os esforços aplicados no desenvolvimento de um *software* é possibilitar seu reaproveitamento na solução de problemas semelhantes. Este reaproveitamento torna-se possível se o sistema é concebido e desenvolvido para ser flexível, sendo capaz de responder a novos requisitos quando utilizado para o desenvolvimento de outros sistemas.

Através da análise de um conjunto de sistemas aplicados na solução de problemas semelhantes verifica-se que as decisões de baixo nível adotadas podem ser bastante semelhantes. É possível abstrair um núcleo comum de serviços no qual a

funcionalidade básica e similar destes sistemas pode ser agrupada. A partir deste núcleo funcional, as diferenças de soluções adotadas por cada aplicação estariam limitadas a conjuntos de dados, ou de funcionalidades, específicos, que poderiam ser acopladas ao núcleo. Assim, desenvolver este núcleo, onde a maioria das decisões de baixo nível pode ser concentrada e reutilizada, pode ser uma solução para o reaproveitamento de esforços no desenvolvimento de aplicações dentro de uma área específica.

Como os esforços de desenvolvimento do núcleo funcional serão reaproveitados por várias aplicações, justifica-se aplicar no mesmo, recursos tais que permitam gerar uma base robusta e de qualidade comprovada. Tendo esta base estruturada como ponto de partida, o desenvolvimento das aplicações passa a se concentrar na busca por soluções adequadas às necessidades e particularidades de cada problema a ser resolvido. Esta é, em termos gerais, a filosofia envolvida na tecnologia de desenvolvimento dos *frameworks* de aplicações.

Framework é "um projeto reutilizável de um sistema que descreve como o sistema é decomposto em um conjunto de objetos interativos. Algumas vezes o sistema é uma aplicação completa; outras ela é apenas um sub-sistema. O *framework* descreve tanto os objetos componentes como a forma de interação entre os objetos. Ele decreve a interface de cada objeto e o fluxo de controle entre os objetos. Ele descreve como as responsabilidades do sistema são mapeadas em seus objetos." (JONHSON e FOOTE 1988[17], WIRFS BROCK 1990 [18], *aput* FAYAD *et al,* 1999).

---

[17] Designing reusable classes. Journal of Object Oriented Programming, 1 (5) June/July, 1988:22-35

[18] Surveying current research in object oriented design. Communications of the ACM 33 (9): 104-124, 1990.

Para MARKIEWICZ e LUCENA (2001) *framework*s são geradores de aplicativos. São sistemas semicompletos reutilizáveis, aplicados no desenvolvimento de *software* específicos a um domínio. Estes sistemas são capazes de gerar aplicativos para todo o conjunto de famílias de problemas correlatos, possuindo pontos de flexibilidade onde adaptações para funcionalidades específicas devem ser feitas para cada aplicação.

Segundo FAYAD e SCHMIDT (1997), *frameworks* de aplicações orientados a objetos representam uma tecnologia promissora para a implementação de aplicativos baseados em projetos de *software* de qualidade comprovada, permitindo reduzir seu custo de implementação, sem perder em qualidade.

Para BUSCCHMANN *et al* (2001) *framework* é um subsistema de *software* parcialmente completo que deve ser instanciado. Ele define a arquitetura para uma família de subsistemas, provendo os blocos de construção básicos para criá-la.

Definido em termos de sua estrutura, um *framework* é um projeto reutilizável de todas as partes de um sistema, que é representado por um conjunto de classes abstratas (e concretas), e pelo modo de suas instâncias interagir. No que se refere ao seu propósito, ele é o esqueleto que pode ser utilizado no desenvolvimento de várias aplicações, customizadas e especializadas por um desenvolvedor ao definir funcionalidades específicas acionadas pelo núcleo central. Assim, aplicações construídas com o uso de *framework*s têm estruturas similares. Comparado a sistemas desenvolvidos com o uso de *toolkits*[19] esta afirmação não é verdadeira, pois, ao utilizar um *toolkit* o desenvolvedor escreve o programa principal da aplicação que chama o

[19] *Toolkit* é um conjunto de classes reutilizáveis que implementam determinadas funções gerais que podem ser redefinidas para gerar resultados específicos.

código que quer reutilizar. Portanto os *toolkits* não impõem um projeto específico a uma aplicação.

*Framework*s são construídos para flexibilidade e para generalidade, cobrindo um domínio inteiro ao invés de problemas particulares. Um *framework* captura as decisões de projeto comuns ao seu domínio, deixando para os desenvolvedores dos aplicativos específicos a definição dos pontos de flexibilização, sem ter de se preocupar com a arquitetura do projeto, mas dedicando-se aos pontos específicos de sua aplicação. O *framework* requer um desenvolvimento mais complexo e dispendioso do que aquele de aplicações diretas. Desenvolvê-lo torna-se, portanto, uma opção válida quando possuímos um conjunto de aplicações semelhantes dentro do domínio que passam a ser produzidas e distribuídas mais rápida e robustamente a partir de uma arquitetura pré-definida. Verificando-se sistemas semelhantes, que abordam os requisitos estabelecidos pelo cliente, pode-se avaliar se cada um destes sistemas poderiam ser instâncias de um *framework* e se valeria à pena desenvolvê-lo.

4.1.1 "Hot spots" X "frozen spots"

Frozen spot (pontos imutáveis) constituem o núcleo reutilizável do *framework* e definem a arquitetura global de um sistema de *software*: seus componentes e os relacionamentos entre eles. São partes do código que já foram implementadas e que são reutilizadas por todas as aplicações desenvolvidas.

Os *hot spots* representam aquelas partes do *framework* que são próprias de sistemas de *software* específicos. São os pontos de flexibilização, formados por classes abstratas e suas interfaces que devem ser implementadas posteriormente na elaboração do aplicativo, uma vez que *framework*s não são executáveis. A classe abstrata provê métodos abstratos ou métodos *template* (descritos por GAMMA et al, 2000) para serem refinados nas subclasses separadamente, de modo que este refinamento não altere a estrutura de base, nem o fluxo geral definido pelo *frozen spot*.

Após a implementação dos *hot spots*, o *framework* utiliza-os através de *callback*, ou seja, o núcleo do *framework* faz chamadas ao código definido no sistema

especializado, ligado a aquele através dos *hot spots,* quando um evento ocorre. São os segmentos de código já implementados que chamam os *hot spots*. Há, então, uma inversão de controle entre a aplicação e o *framework* sobre o qual ela está baseada.

Frameworks que permitem o desenvolvimento de aplicações basicamente através da composição de novos objetos, a partir de classes concretas já disponibilizadas, são chamados de caixa preta (*black box*). Esta denominação deve-se ao fato de que, através da composição, a montagem de objetos pode ser realizada sem que o usuário tenha de conhecer o interior das classes já existentes Em contraposição, *frameworks* que permitem o desenvolvimento de aplicações predominantemente através de herança, tornando o interior da classe base visível para as subclasses, recebem a denominação caixa branca (*white box*). Segundo este critério, se um *framework* utiliza os dois tipos de reutilização, ele é classificado como caixa cinza (*grey box*).

Frameworks caixa preta disponibilizam a forma mais simples de implementação de aplicações através da conexão de componentes existentes. A definição de novas subclasses concretas através de *framework*s caixa branca representa um caminho mais complicado para a implementação da aplicação, pois exige do desenvolvedor um maior conhecimento da estrutura interna do *framework*. Porém, a alternativa mais complexa refere-se ao desenvolvimento de aplicações através de alteração das classes abstratas do *framework*, adicionando variáveis e operações a estas.

4.1.2 Desenvolvimento de *Frameworks*:

MARKIEWICZ e LUCENA (2001) fazem um paralelo entre o desenvolvimento de sistemas orientados a objetos e o desenvolvimento de *frameworks.* Segundo estes autores as fases do desenvolvimento de sistemas orientados a objetos são: análise do problema, projeto, construção, transição, e instanciação (FIGURA 4.1). Uma definição semelhante de etapas é apresentada por PAULA FILHO (2000) para o PRAXIS (Processo para aplicativos extensíveis Interativos). PAULA FILHO define as seguintes etapas: Requisitos, análise, desenho, testes e implementação. Para o desenvolvimento de *framework*s MARKIEWICZ e LUCENA (2001) destacam três estágios principais:

análise de domínio, projeto do *framework* e a instanciação do *framework* (FIGURA 4.2).

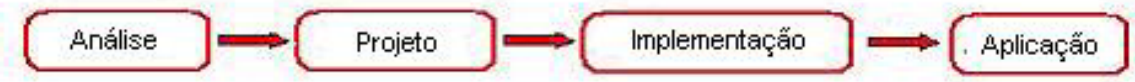

FIGURA 4.1 – Fases de desenvolvimento para sistemas orientados a objetos.

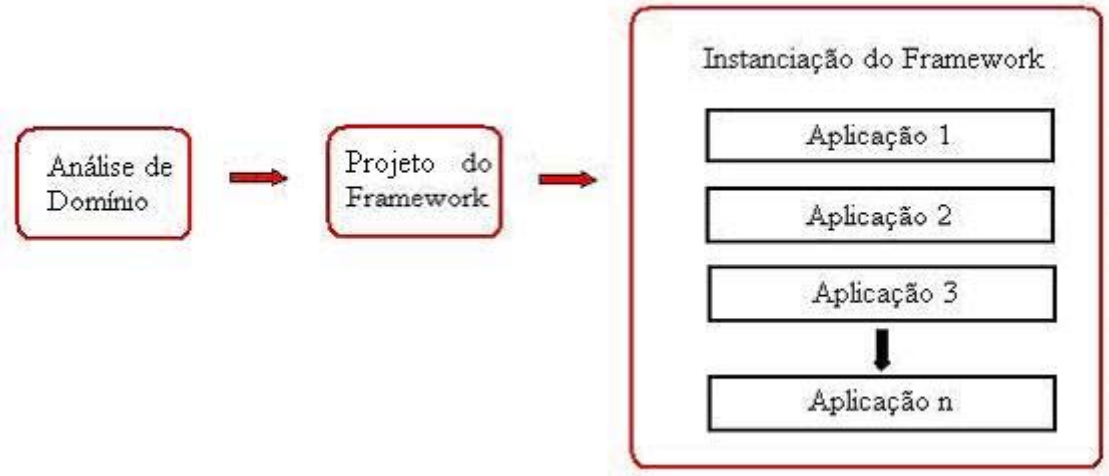

FIGURA 4.2 – Processo de desenvolvimento de *frameworks*.

No desenvolvimento tradicional, a análise restringe-se a um problema específico e o aplicativo gerado fica associado somente a este problema. No desenvolvimento de *frameworks* a fase de análise levanta os requisitos de todo um domínio, além de possíveis requisitos futuros. O levantamento destes requisitos pode ser feito através de experiências pessoais e publicadas, programas similares existentes e modelos ou protótipos. Esta fase busca caracterizar o tamanho e a complexidade de um dado domínio. Domínios extensos e complexos exigem um tempo muito grande de análise e de desenvolvimento, enquanto domínios muito pequenos representam aplicativos muito semelhantes para justificar o desenvolvimento de *framework*s. Durante a análise de domínio os *hot spots* e os *frozen spots* são identificados.

Após a identificação do domínio, conhecida sua extensão e definidos os possíveis *frozen spots* e *hot spots*, deve-se avaliar a viabilidade de desenvolver um *framework* ao invés de uma aplicação tradicional. Para isso devem ser consideradas as necessidades do problema, o que tem sido desenvolvido na área em questão, a possibilidade de criação de vários aplicativos para este domínio e quais os custos e benefícios envolvidos em tal desenvolvimento. Deve-se, portanto, avaliar as implicações deste desenvolvimento como: necessidade de pesquisas, de parcerias, mudanças de filosofia e de investimento, além de levantar a equipe necessária para o desenvolvimento. As expectativas relacionadas com o desenvolvimento do *framework* também devem ser listadas e avaliadas como: inovação, desenvolvimento de aplicações personalizadas mais rápidas e robustas, atendimento de maior público, etc.

Após a análise, na fase de projeto, define-se a arquitetura do sistema, os *hot spots* são definidos e os *frozen spots* são implementados. Nesta etapa a extensibilidade e a flexibilidade proposta na análise de domínio são delineadas.

Na fase de instanciação do *framework* os *hot spots* são implementados correspondendo ao desenvolvimento de aplicativos específicos. Vários aplicativos podem ser gerados a partir do *framework*. Esta fase engloba a construção e transição do desenvolvimento tradicional. Cada sistema gerado reutilizará os *frozen spots* do *framework* e, portanto, uma aplicação desenvolvida a partir de um *framework* deve ajustar-se a este. O projeto da aplicação deve começar com o projeto do *framework*.

Uma aplicação desenvolvida com o uso de *frameworks* apresenta três partes:

1. O *framework* definindo a arquitetura do sistema,
2. Subclasses concretas, derivadas das classes do *framework* formando os *hot spots,*
3. Demais objetos e relacionamentos que não têm relação com o *framework.*

O desenvolvimento de *framework*s requer, ainda, um processo iterativo através do qual sua adequação ao domínio é aprimorada, juntamente com sua estrutura e sua funcionalidade. Em uma primeira versão geralmente o *framework* originário é do tipo

caixa branca. A cada nova implementação o *framework* é aprimorado. Ao longo desse processo um *framework* caixa branca pode ser transformado em um *framework* caixa preta

Outro fator que deve ser considerado durante o desenvolvimento de um *framework* é a possibilidade de sua integração com outros *frameworks* existentes. Neste caso o comportamento não coerente, a superposição de domínios, intenções de projeto concorrentes, a falta de acesso ao código fonte e a ausência de métodos ou padrões para os *frameworks* integrados podem gerar grandes dificuldades de desenvolvimento (MATTSON *et al,* 1999).

4.2 Padrões

Segundo o critério de reutilização, um *software* desenvolvido para auxiliar na solução de um problema deve ser específico para o mesmo, mas genérico o bastante para atender futuros requisitos. Assim, um *software* não deve ser estanque, mas adaptável a futuros requisitos. Por outro lado, o desenvolvimento de projetos flexíveis não é uma tarefa simples. Construir projetos flexíveis e reutilizáveis é uma tarefa mais dispendiosa que a produção de projetos específicos e não adaptáveis a novas implementações.

O paradigma da POO apresenta como vantagens em relação à programação estruturada: a reutilização de código, a modulação de componentes e, portanto, a redução do ciclo de desenvolvimento de programas (DEVLOO, 1997). Em sistemas orientados a objetos o funcionamento geral do sistema é definido pelo comportamento dos objetos (BRAVO *et al,* 2007). Assim, para projetar um *software* reutilizável, deve-se abstrair do mundo real objetos que possam representar o problema a ser solucionado no ambiente virtual, permitindo que a estrutura final adotada e o relacionamento entre

os objetos definidos, permitam a reutilização destes em problemas semelhantes (GAMMA *et al*, 2000).

Quando se encontra uma boa solução para um problema é desejável que ela possa ser reaproveitada em problemas correlacionados e uma solução utilizada repetidamente, pode ser modelada em um padrão de projeto: um conjunto de classes com papéis e comportamentos bem definidos dentro de um relacionamento. Sendo identificado um padrão de projeto, este pode ajudar a escolher alternativas que facilitam a construção de novos *softwares* que busquem solucionar problemas semelhantes.

Padrões foram definidos originalmente pelo arquiteto Christopher Alexander[20] ao afirmar que: um padrão descreve um problema relacionado a um determinado ambiente de convívio e a forma como ele é solucionado, de tal forma que a solução pode ser utilizada várias vezes, porém sem nunca ser simplesmente repetida. Seu trabalho é baseado na observação de organizações espaciais em arquitetura e na busca de padrões capazes de representar soluções aplicadas em contextos semelhantes, propondo gabaritos para a descrição dos padrões. Um padrão catalogado pode ser aplicado no desenvolvimento de um novo projeto dentro de seu contexto de aplicação. O uso de um conjunto de padrões para o desenvolvimento de projetos arquitetônicos pode tornar o processo de projeto determinístico e capaz de ser repetido. (ALEXANDER *apud* GAMMA *et al*, 2000).

Aplicado no desenvolvimento de sistemas de *software*, segundo SCHMIDT *et al* (1996) e COPLIEN (2003), padrões descrevem soluções bem sucedidas para problemas comuns de *software*, representando soluções recorrentes para estes problemas.

---

[20] ALEXANDER Christopher: *A Pattern Language*. Oxford University Press, New York, 1977.

Segundo BUSCCHMANN *et al* (1996) um padrão aponta para um problema de projeto que surge em uma situação específica, apresentando uma solução para o mesmo e proporcionando o desenvolvimento de sistemas com propriedades definidas. Padrões não relatam novidades, mas experiências bem sucedidas e quanto mais um padrão é usado com sucesso, mais importante e valioso se torna. Eles documentam experiências e conhecimentos adquiridos a partir de projetos existentes e comprovados, identificando e especificando abstrações que estão acima do nível de simples classes e instâncias ou de componentes. Um padrão descreve vários componentes, classes ou objetos, e detalha suas responsabilidade e relacionamentos, assim como sua cooperação.

Além de contribuir para a solução de um problema, um padrão de projeto pode melhorar a documentação e a manutenção de sistemas ao fornecer especificações claras sobre os relacionamentos entre classes e objetos. Eles promovem um vocabulário comum para o entendimento dos princípios de um projeto, facilitando discussões de problemas de projeto e suas soluções. Eles também descrevem a intenção por traz do projeto de um sistema, contribuindo para que outros desenvolvedores não violem a mesma, ao estender ou modificar a arquitetura original, ou ao modificar o código de um sistema.

Um padrão não detalha completamente uma solução. Ele disponibiliza um esquema para uma solução genérica a uma família de problemas, ajudando a gerenciar a complexidade de um sistema, e deve ser implementado a cada novo desenvolvimento.

Apesar de padrões não serem métodos ou processos de desenvolvimento de *software*, eles complementam aqueles existentes. Eles ajudam a fazer a ligação entre as abstrações das fases de análise de domínio e do projeto de arquitetura, com a concretização destas nas fases de implementação e manutenção. Nas fases de análise e de projeto os padrões auxiliam na seleção de arquiteturas bem sucedidas. Nas fases de implementação e manutenção eles ajudam a documentar as propriedades estratégicas do sistema em um nível mais alto do que o de códigos fonte e de modelos.

Geralmente os padrões não trabalham sozinhos, eles podem ser combinados de maneira que a contribuição de todos forme um processo que gerencie uma solução

ordenada do desenvolvimento de *software*. Quando a combinação de padrões cooperantes cobre um domínio particular, ela forma uma Linguagem de Padrões.

4.2.1 Contexto, Problema e Solução:

Segundo BUSCHMANN *et al* (1996), todo padrão é fundamentado em um esquema que compreende três partes: um contexto, um problema e uma solução.

O contexto é uma situação que dá origem a um problema. Ele descreve situações, nas quais o problema ocorre, devendo ser generalista.

O problema descreve uma questão que surge repetidamente em um dado contexto. O termo "força" é utilizado para denotar qualquer aspecto do problema que deva ser considerado na solução do mesmo como: requisitos que a solução deva satisfazer, condições que devam ser consideradas e propriedades desejáveis que a solução deva apresentar.

A solução mostra como resolver o problema recorrente, como balancear as "forças" associadas ao mesmo. Toda solução especifica uma determinada estrutura, ou configuração especial dos elementos, e um comportamento de tempo de execução.

Após a aplicação de um padrão, uma arquitetura apresenta uma estrutura particular que dá sustentação ao papel especificado pelo padrão, porém ajustada e adaptada às necessidades específicas do problema em questão.

4.2.2 *Frameworks* x Padrões:

O desenvolvimento de *framework*s freqüentemente utiliza padrões de projeto que auxiliam em sua estruturação e instanciação. Um *framework* que se beneficia de padrões de projeto apresenta um maior grau de reutilização, tanto por respeitar

determinadas normas de instanciação e de relacionamento entre seus elementos, quanto por ter sua documentação facilitada pelos padrões utilizados.

Sob o ponto de vista de *framework*s, padrões podem ser vistos como seus blocos construtivos. Sob o ponto de vista dos padrões, um *framework* pode ser visto como um padrão para sistemas de *software* completos em um dado domínio de aplicação.

As diferenças principais entre padrões de projeto e *framework*s são:

- Padrões de projeto são mais abstratos do que *framework*s, pois estes últimos possuem uma implementação central, enquanto aqueles possuem apenas uma especificação de aplicação de uso e de relacionamentos.
- Padrões de projeto são elementos de arquitetura menores que *framework*s, estes últimos podem ter vários padrões de projeto em sua constituição.
- Padrões de projeto são menos especializados que *framework*s não tendo um domínio particular de aplicação.

4.2.3 Categorias de padrões:

Os padrões existem em vários graus de escala e de abstração, cobrindo diferentes e importantes áreas do desenvolvimento de *software*. Alguns padrões ajudam a estruturar um sistema em subsistemas, outros suportam o refinamento de subsistemas e de componentes ou das relações entre eles. Há ainda aqueles padrões que ajudam a implementar aspectos particulares de projeto em uma linguagem de programação específica. Assim, segundo BUSCCHMANN *et al* (1996), os padrões podem ser agrupados em três categorias: padrões de arquitetura, padrões de projeto e idiomas.

Os padrões de arquitetura tratam da organização de estruturas de sistemas. Eles proporcionam um conjunto predefinido de subsistemas com responsabilidades especificadas por regras e diretrizes, as quais organizam o relacionamento entre eles. Os padrões de projeto proporcionam um esquema para o refinamento de subsistemas. Eles descrevem uma estrutura de componentes comunicantes que solucionam um problema

geral de projeto. São padrões de média escala, sendo menores do que os padrões de arquitetura, mas independentes de uma linguagem de programação ou paradigma de programação. Um idioma é um padrão de baixo nível, específico a uma linguagem de programação.

4.2.4 Padrões de Arquitetura

Os padrões de arquitetura de *software* tratam da especificação da estrutura de uma aplicação. BUSCHMANN *et al* (1996) descreve vários padrões de arquitetura divididos nas categorias: padrões de estruturação, padrões para sistemas distribuídos, padrões para sistemas de interação e padrões para sistemas adaptáveis.

Os padrões de estruturação são padrões que proporcionam a definição da arquitetura do sistema definindo uma subdivisão de alto nível do sistema, formada por partes constituintes. Eles promovem uma decomposição controlada de uma funcionalidade geral de um sistema em tarefas cooperantes. Os padrões para sistemas distribuídos proporcionam uma infra-estrutura completa para aplicações distribuídas. Padrões para sistemas de interação são aqueles que permitem um elevado grau de interação homem x computador com o uso de interfaces de usuário. Padrões para sistemas adaptáveis devem proporcionar uma extensão das aplicações e a adaptação destas à evolução de tecnologias e à mudança de requisitos funcionais, sem afetar seu núcleo de funcionalidade ou suas abstrações principais de projeto.

A escolha de um padrão de arquitetura deve ser guiada pelas propriedades gerais da aplicação a ser desenvolvida. Antes de escolher um padrão específico, deve-se explorar várias alternativas, pois diferentes padrões de arquitetura implicam em diferentes estruturas e conseqüentemente diferentes sistemas. Por outro lado, sistemas que devem suportar diferentes requisitos, muitas vezes não podem ser totalmente estruturados por apenas um padrão específico de arquitetura, sendo necessária a combinação de dois ou mais padrões ou a utilização de características destes padrões para gerar a arquitetura mais adequada ao sistema. Além disso, a arquitetura completa de um sistema não é representada apenas pelo padrão, ou combinação de padrões

adotada. Os padrões apenas contribuem para definir a estrutura de um sistema. Esta deve ser mais refinada e integrada à funcionalidade da aplicação, através do detalhamento de componentes e de relacionamentos. Este refinamento pode ser auxiliado por outra categoria de padrões: os padrões de projeto e os idiomas.

Para o desenvolvimento da arquitetura do *framework* REMFrame foram escolhidos os padrões MVC (*Model-View-Controller)* e *Microkernel* descritos por BUSCHMANN *et al* (1996), tendo em vista as características principais às quais ele deve responder: flexibilidade da interação com o usuário e integração entre diferentes sistemas. Os conceitos oriundos do padrão MVC permitiram organizar a estrutura do sistema para flexibilização segundo a ótica de sistema interativo, enquanto o padrão *Microkernel* permitiu estruturar o sistema para flexibilização e extensão segundo a ótica de sistema adaptável. Estes dois padrões serão apresentados nos subitens adiante.

Os demais padrões de arquitetura, descritos por BUSCHMANN *et al* (1996), são listados abaixo, juntamente com uma breve descrição do domínio de aplicação de cada um. Em seguida são discutidos alguns padrões e os motivos pelos quais eles não foram adotados no desenvolvimento do *framework* REMFrame.

Layer: é um padrão de organização que visa estruturar aplicações que podem ser decompostas em grupos de sub-rotinas, onde cada grupo está localizado em um determinado nível de abstração.

Pipes e Filters: padrão aplicado na estruturação de sistemas que processam um fluxo de dados. O sistema é dividido em estágios seqüenciais e cada etapa de processamento é encapsulada por um componente de filtro. Os dados são passados pelos condutores entre filtros adjacentes.

Blackboard: é um padrão útil para problemas nos quais não há uma estratégia conhecida para a determinação da solução. Uma solução parcial ou aproximada é construída, então pela atuação conjunta de vários subsistemas especialistas.

Broker: Padrão direcionado para a estruturação de sistemas distribuídos com componentes não acoplados, os quais interagem por chamadas de serviço remotas.

Document-View: Aplicável quando a apresentação visual e a manipulação de eventos são bem interligadas, porém limita a variação de controladores.

Presentation-Abstraction-Control: Define a estrutura de um sistema interativo na forma de uma hierarquia de agentes cooperantes. Cada agente é responsável por um aspecto específico da funcionalidade da aplicação e é constituído por três componentes: *presentation*, *abstraction* e *control.*

Reflection: Provê mecanismos para a troca da estrutura e do comportamento de sistemas dinamicamente. Suporta a modificação de aspectos fundamentais como tipos de estruturas e mecanismos de chamadas de funções.

Avaliando-se os domínios de aplicação de cada padrão apresentado, verificou-se que: os padrões *Broker*, *Blackboard* e *Reflection* apresentam domínios distintos daquele definido para o *framework* REMFrame, não sendo, portanto, aplicáveis a este. Segundo BUSCHMANN *et al* (1996), o padrão *Pipes e Filters* também não seria aplicável porque um sistema interativo, onde cada etapa da execução depende de ações externas, não é divisível em estágios seqüenciais. Foram avaliados, então, os padrões *Layers*, *Document-View* e *Presentation-Abstraction-Control*.

Os padrões *Document-View* e *Presentation-Abstraction-Control*, assim como o padrão MVC, organizam o sistema sob o ponto de vista de sua interatividade.

O *Document-View* é uma variação do padrão MVC, combinando no módulo *View* as responsabilidades dos módulos *view* e *controller* do padrão MVC. Já o componente *document* corresponde ao *model* no MVC. Isto é útil principalmente em plataformas de interface gráfica de usuários (GUI[21]) onde a apresentação visual e a

[21] GUI - *Grafical Users Interface*

manipulação dos eventos são bem interligadas. A variante *Document-View* é utilizada no ambiente do Visual C++, no *framework Microsoft Foundation Classes* (MFC).

Este tipo de organização limita, porém, a variação dos *controllers*, condicionando esta à variação da visualização. A utilização deste padrão no desenvolvimento do REMFrame implicaria em um forte acoplamento entre visualização e controle, contrariando três dos pré-requisitos estabelecidos para o sistema: permitir o desenvolvimento dos sistemas clientes através do uso de diferentes plataformas gráficas, permitindo diferentes modos de interação com o usuário, assim como possibilitar integração entre diferentes sistemas CAE.

O padrão *Presentation-Abstraction-Control* descreve um sistema interativo estruturado em agentes de três níveis: os *top agents*, os *intermediate agents* e os *bottom agents*. Cada agente possui os componentes: controle, abstração e apresentação, e pode representar um elemento que armazena informação ou controla um processamento. Um agente pode ser responsável por receber e mostrar dados, manter os dados do modelo ou fazer a comunicação com outros sistemas, etc.

Os autores BUSCHMANN *et al* (1996) salientam que a existência de agentes independentes (com dados, comportamento e visualização próprios) permite um tratamento diferenciado para cada agente. Por outro lado esta estruturação acarreta em um aumento considerável da complexidade do sistema por aumentar o número de classes (cada elemento tendo três componentes). Assim, sistemas nos quais cada objeto individual do modelo é representado por um agente apresentam estruturas muito refinadas tornando a manutenção do sistema difícil. Em comparação ao padrão MVC, padrão *Presentation-Abstraction-Control* gera um sistema onde a obrigatoriedade de tratar separadamente uma visualização para cada elemento torna o desenvolvimento do sistema gráfico um trabalho muito complexo se comparado ao MVC onde um elemento de visualização pode ser desenhado de forma a representar elementos do modelo que sempre se apresentam em conjunto.

LEITE E FRANCO (2005) fazem uma abordagem mais detalhada sobre a aplicabilidades dos padrões MVC, *Document-View* e *Presentation-Abstraction-Control*

no desenvolvimento de um módulo gráfico para a modelagem de estruturas espaciais reticuladas, gerado a partir do *framework* REMFrame.

O padrão *Layers*, consiste na estruturação do sistema em camadas superpostas bem definidas, onde cada camada implementa um conjunto de serviços e comunica-se com as camadas adjacentes, transferindo solicitações ou dados. O *framework* REMFrame trata um processo dividido em etapas bem definidas (modelagem, análise, dimensionamento, detalhamento e orçamento) e implementadas por diferentes componentes. Cada uma destas etapas pode ser entendida como uma camada e as aplicações finais obtidas pela integração de diferentes etapas do processo seriam formadas por camadas, porém sem a característica de superposição, mas sim interagindo segundo um fluxo radial de dados onde o núcleo do *framework* fica responsável por coordenar a interação entre os sistemas (periféricos) independentes. Da mesma forma o núcleo do *framework* apresenta módulos independentes segundo a mesma organização, onde um módulo central faz chamadas a vários outros módulos auxiliares. Desta forma, este padrão não foi adotado na definição da arquitetura do *framework*.

Passamos então a uma análise mais detalhada dos padrões MVC e *Microkernel.*

O padrão Model-View-Controller – MVC

O padrão *Model-View-Controller* (MVC) é um padrão de arquitetura para sistemas de interação que divide uma aplicação interativa em três componentes: *model*, *view* e *controller*. O *model* (modelo) encapsula o núcleo de dados e de funcionalidade da aplicação, o *view* (visualização) apresenta informações oriundas do modelo para o usuário e o *controller* (controlador) manuseia as informações segundo eventos acionados pelo usuário (BURBECK, 2003). *Views* e *controllers* juntos formam a interface de usuário.

O domínio de aplicação do MVC é o de aplicações interativas com interface flexível homem x computador, onde é permitida a variação das interfaces de usuário. A solução do problema é influenciada pelas seguintes forças:

- Deve-se permitir a apresentação de uma informação por diferentes formas de visualização,
- A visualização e o comportamento da aplicação devem refletir imediatamente alterações realizadas sobre os dados tratados,
- Mudanças na interface de usuário devem ser facilmente implementadas,
- A substituição da interface de usuário não deve afetar o código do núcleo da aplicação.

O componente *model* contém o núcleo funcional da aplicação. Ele encapsula dados e disponibiliza procedimentos que executam o processamento destes dados. Estes procedimentos são requisitados pelos *controllers* que interpretam e repassam as solicitações do usuário. O componente *model* é independente da representação dos dados ou da entrada de dados.

Segundo este padrão cada *view* apresenta um componente *controller* associado a ele e geralmente apresenta métodos que permitem ao *controller* manipular a visualização dos dados. Os *controllers* recebem a entrada de dados através de eventos e cada evento é traduzido em solicitação de serviços para o *model* ou para o *view*.

Aplicação do padrão MVC ao framework REMFrame

No desenvolvimento do *framework* REMFrame, o padrão MVC foi adotado com algumas adaptações para melhor responder às características do domínio abordado. Ele foi adotado como um padrão que define o comportamento geral do sistema frente a alterações dos dados e sua visualização. Desta forma as classes são classificadas em classes de dados, de representação ou de controle sem, porém, estarem inseridas nos três componentes preconizados pelo padrão.

Foram criadas visualizações para elementos do modelo encapsulados por classes específicas como: barras, seções transversais e carregamentos. Outras visualizações representam um determinado conjunto de dados relacionados a um ou mais elementos

do modelo estrutural e armazenados por classes contêineres. Neste caso temos: as restrições nodais, as liberações aplicadas aos elementos de análise, a deformada da estrutura e os diagramas de esforços solicitantes.

Devido à necessidade de um criterioso controle de dados para garantir uma constante consistência entre todos os elementos da estrutura representada, o relacionamento entre os objetos representantes dos módulos básicos (modelo, visualização e controle) sofreu pequenas alterações. Na estrutura desenvolvida não foi definido um controlador para cada elemento de visualização, mas um controlador que gerencia a adição, a subtração ou a atualização de dados no modelo estrutural como um todo (classe *ProjectController*), fazendo o papel de fachada para o sistema, semelhante ao padrão *Façade* (GAMMA *et al*, 2000). Este controlador manipula alguns dados de entrada, repassando-os para controladores específicos dos diversos pacotes ou componentes envolvidos no processo (classes *Project*, *Model* e *LoadCase*). Estes controladores específicos funcionam ainda como classes de banco de dados, armazenando, em listas ou dicionários, os elementos de modelo que gerenciam. Eles manipulam e repassam dados e solicitações para as classes de modelo, podendo criar e inserir elementos destas classes em suas estruturas de armazenamento. Quando elementos do modelo são criados, estes controladores promovem a vinculação entre modelo e visualização adicionando ao elemento do modelo (derivado da classe base *Subject*) um elemento de visualização (derivado da classe base *View*). Nesta vinculação entre modelo e visualização, foi utilizado o padrão de projeto *Abstract Factory* (GAMMA *et al*, 2000) através da classe abstrata *ViewFactory*, para permitir que cada aplicação gerada defina o conjunto de elementos de visualização a ser utilizado. Esta classe constitui um dos *Hot spots* do *framework*.

A estrutura de classes adotada para o REMFrame será estudada com mais detalhes no capítulo 5.

No padrão MVC a comunicação entre os componentes *model, view* e *controller* garante uma interação coerente com o usuário, permitindo uma constante atualização da visualização dos dados do modelo. Na estrutura adotada para o *framework* REMFrame um elemento de modelo mantém uma lista dos elementos de visualização relacionados a ele. Sempre que seus dados são alterados, uma notificação é gerada para cada elemento

de visualização a ele associado. Os elementos de visualização recuperam os novos dados do modelo e atualizam sua representação. Este mecanismo de propagação de mudanças é implementado através do padrão de projeto *Observer* (GAMMA *et al*, 2000). O mecanismo de propagação de mudanças é a única ligação entre os elementos de dados, os objetos de visualização (classes derivadas de *View*) e os controladores (classes de banco de dados do pacote *DataBase Classes* e *ProjectController*).

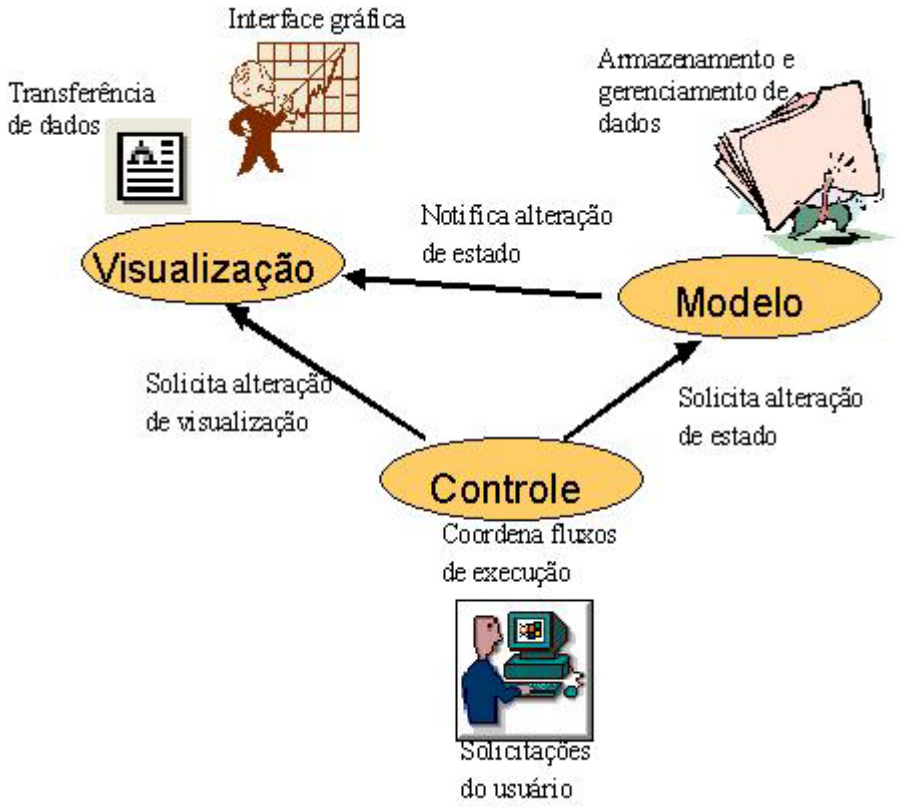

FIGURA 4.3 - Componentes básicos do padrão MVC x Mecanismo de propagação de mudanças.

A FIGURA 4.3 apresenta o relacionamento geral entre os diferentes tipos de objetos adotado na implementação da arquitetura MVC no *framework* REMFrame. Nessa figura os objetos de dados são representados pelo módulo *Model*, os elementos de visualização são representados pelo módulo *View*, e os controladores, pelo módulo *Controller*.

No desenvolvimento do *framework* foram definidas três classes base, cada uma representando um dos módulos do padrão MVC. A classe *Subject* é a classe base das classes de modelo (dados) que podem ser apresentadas para o usuário através de uma representação gráfica. As classes *View* e *Controller* compartilham uma classe base comum (*Observer*) que define a interface de atualizações. Uma quarta classe denominada *Componente* reúne funcionalidades gerais compartilhadas por várias classes de modelo e de controle podendo ser utilizada na definição das classes derivadas destas duas classes. A estrutura de classes que implementam a arquitetura MVC foi organizada no componente *ClassesMVC* do *framework* (FIGURA 5.2, item 5.1.1).

A aplicação do padrão MVC na estruturação do *framework* REMFrame para geração de sistemas de modelagem de estruturas reticuladas possibilitou a definição de um importante *hot spot* representado pela as classe *View*. Além disso, a arquitetura MVC, permitiu ao *framework* responder aos seguintes requisitos:

- Possibilidade de visualização de diferentes propriedades aplicadas a uma mesma estrutura reticulada através de vários modelos gráficos: unifilar, geométrico sólido, carregamentos aplicados, análise, deformada da estrutura, diagramas de esforços, etc.,
- Compatibilidade entre diferentes visualizações de um modelo estrutural,
- Possibilidade de alteração dos objetos *view* e *controller* em tempo de execução,
- Permitir o desenvolvimento de aplicações em diferentes plataformas gráficas sem a alteração do núcleo de funcionalidade proposto.

O padrão *Microkernel*:

O padrão *Microkernel* é um padrão para sistemas adaptáveis, capazes de aceitar mudanças de requisitos. Ele prevê a separação de um núcleo funcional mínimo, chamado *microkernel*, de outros módulos do sistema. Estes outros módulos podem ser módulos internos, que tratam funcionalidades estendidas do sistema, ou sistemas

externos especificados por clientes. O núcleo funcional mínimo opera como um conector ao qual podemos adicionar extensões de serviços do sistema, coordenando as colaborações entre os diversos sistemas conectados. Este padrão permite a definição de um projeto original pequeno e eficiente, suportando sua extensão com adição de novos módulos e serviços.

O padrão *microkernel* define cinco tipos de componentes participantes: servidores internos, servidores externos, adaptadores, clientes e o *microkernel* (FIGURA 4.4).

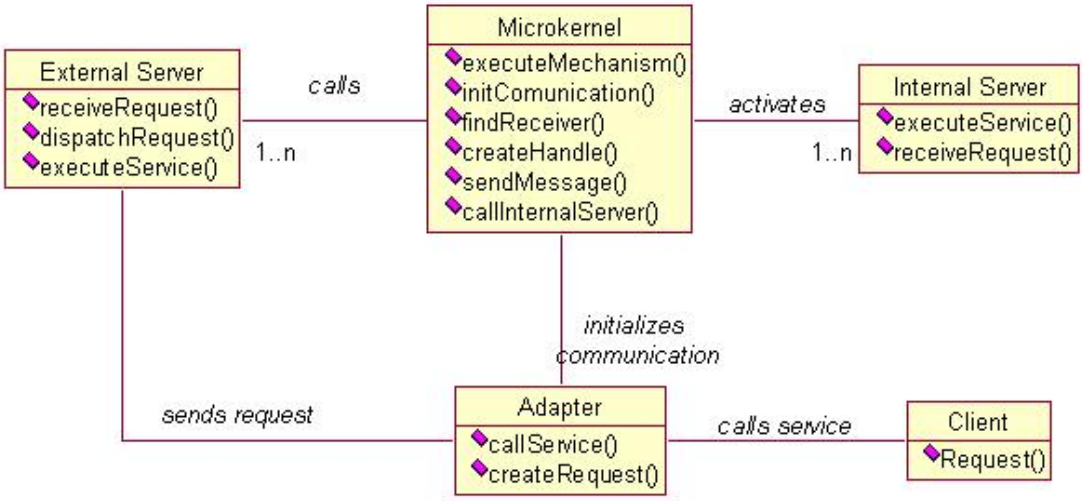

FIGURA 4.4 - Estrutura do padrão *Microkernel* (figura extraída de BUCHMANN *et al*, 1996)

O *microkernel* é o componente principal do padrão sendo responsável por manter os recursos do sistema. Ele disponibiliza um núcleo funcional que permite outros componentes, processados de forma independente, comunicarem-se uns com os outros. Este componente também é responsável por prover uma interface que permite aos outros elementos acessar sua funcionalidade.

Os servidores internos, ou subsistemas, implementam funcionalidades que não poderiam ser implementadas no *microkernel* sem aumentar seu tamanho ou complexidade. Eles estendem a funcionalidade provida pelo *microkernel*, oferecendo

funcionalidades adicionais e serviços complexos. Os servidores internos são ativados ou carregados pelo *microkernel* quando necessário, sendo acessíveis somente por este componente.

Os servidores externos são componentes que utilizam o *microkernel* para implementar diferentes políticas para domínios de aplicações específicos. Eles recebem as solicitações de serviços de aplicações clientes tratando-as e repassando-as para o *microkernel.*

O cliente é uma aplicação associada a um servidor externo. Ele acessa apenas as interfaces de programação providas pelo servidor externo.

Adaptadores representam as interfaces entre clientes e seus servidores externos, permitindo que os clientes acessem os serviços de seus servidores externos de um modo portável. Adaptadores protegem os clientes de detalhes de implementação específicos do *microkernel*. Eles transmitem chamadas de um cliente a um servidor apropriado.

Em termos gerais o padrão *Microkernel* é aplicável a sistemas que possuem vários módulos com funções bem definidas e deseja-se prever a extensão do sistema com novos módulos que possam surgir. Este padrão possibilita, então, a comunicação entre os módulos do sistema, evitando que o módulo que exige um determinado serviço tenha de conhecer todos os módulos que podem fornecer este serviço. O núcleo central da aplicação controla a interação entre os módulos disponíveis, isolando as pontas do processo.

Aplicação do padrão** Microkernel **ao framework REMFrame

Um dos requisitos estabalecidos para o *framework* desenvolvido é que esse fosse capaz de integrar diferentes sistemas adotados em etapas bem definidas do processo produtivo de estruturas reticuladas, ou seja, tratar em um único fluxo os dados provenientes das etapas de modelagem, análise, dimensionamento, detalhamento e orçamento. Além disso, um sistema pode utilizar vários módulos para desenvolver uma mesma função como, por exemplo: utilizar diferentes sistemas de análise, ou diferentes

sistemas de dimensionamento, ou ainda, diferentes plataformas gráficas. Assim vários módulos podem estar disponíveis ou podem ser adicionados no futuro.

A utilização de alguns conceitos e do padrão *Microkernel* na definição da arquitetura do *framework* permitiu a este desempenhar o papel de gerenciador de dados e de fluxos de dados, controlando toda a comunicação envolvida no processo. A aplicação gerada é organizada a partir de conjunto predefinido de componentes com responsabilidades e relacionamentos especificados por regras e políticas definidas em módulo central. Este módulo define em tempo de execução qual dos sistemas disponíveis será utilizado a partir da solicitação do usuário, configurando o ambiente ou domínio requerido pelo sistema, e ainda manipulando os dados do modelo para gerar a formatação requerida por aquele.

Procurando aplicar ao *framework* a estrutura proposta pelo padrão *Microkernel* aquele foi organizado em diferentes componentes e pacotes. Funcionalidades específicas foram organizadas em componentes independentes: *MathClasses* para cálculos geométricos, *MaterialClasses* para tratamento de dados de propriedades físicas, *SectionClasses* para tratamento de dados referentes a seções transversais, *FillerClasses* para transferência de dados. Os módulos assim obtidos desempenham o papel de servidores internos. Um módulo central faz o gerenciamento das informações do modelo vinculando dados e informações obtidas destes módulos independentes. Neste módulo estão classes de banco de dados e de controle além dos dados que representam os elementos essenciais de uma representação do modelo estrutural. Este componente central recebeu o nome de *Kernel*. Os servidores externos foram aqui definidos como os sistemas externos a serem integrados, assim como a plataforma gráfica utilizada para gerar as representações do modelo, enquanto os clientes são os sistemas especializados desenvolvidos com o uso do *framework*. A classe *ProjectController* representa o componente *adapter* deste padrão ao realizar a interface entre os componentes clientes, e demais classes do componente *Kernel*.

4.2.5 Padrões de Projeto

Enquanto padrões de arquitetura tratam a organização ou estruturação de um sistema em um conjunto predefinido de componentes com responsabilidades especificadas por regras e diretrizes, existem outros padrões que procuram responder à implementação de um ou mais componentes do sistema. Os padrões de projeto representam padrões de média escala no sistema de padrões apresentado por BUSCCHMANN et al (1996). Eles não afetam a estrutura fundamental de sistemas de *software*, mas têm grande influência na arquitetura de subsistemas.

"Padrões de projeto são descrições de objetos e classes comunicantes que são customizados para resolver um problema geral de projeto em um contexto particular" (GAMMA et al, 2000).

Um padrão de projeto nomeia, abstrai e identifica os aspectos principais de uma estrutura comum para que a mesma seja reutilizada na criação de projetos orientados a objetos. O padrão de projeto identifica as classes e instâncias participantes de uma solução, seus papéis, suas colaborações e a distribuição de responsabilidades entre elas.

GAMMA et al, 2000, classifica os padrões de projeto em padrões de criação, padrões estruturais e padrões comportamentais.

Padrões de Criação

Os padrões de criação abstraem o processo de criação, tornando um sistema independente de como seus objetos são criados, compostos e representados. Estes padrões encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema e ocultam o modo como as instâncias dessas classes são criadas e agrupadas. O que o sistema sabe sobre os objetos é que suas classes são definidas por classes abstratas.

Os padrões de criação podem tornar um sistema mais flexível. Eles facilitam a modificação das classes que definem os componentes de um sistema, fornecendo

diferentes maneiras de remover referências explícitas a classes concretas do código necessário para criá-las.

Abstract Factory:

O padrão *Abstract Factory* (GAMMA et al, 2000) fornece uma interface para a criação de uma família de objetos relacionados ou dependentes, sem especificar suas classes concretas. Este padrão é aplicável quando o sistema deve ser independente de como os produtos são criados, compostos ou representados, ou quando uma família de objetos-produto for projetada para ser utilizada em conjunto e é necessário garantir esta restrição.

O padrão *Abstract Factory* é composto por uma classe que declara uma interface para criação de produtos (FIGURA 4.5). O cliente não conhece a classe concreta que está gerando o produto, mas sim a interface da classe abstrata e, portanto, a interface do tipo de objeto, ou produto, a ser criado.

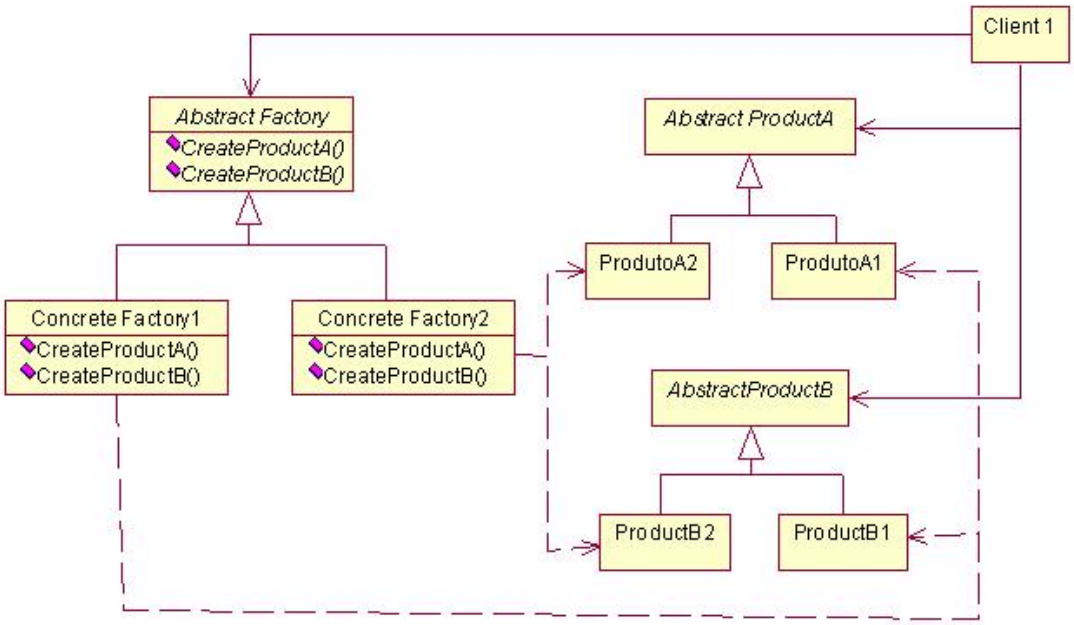

FIGURA 4.5 - Estrutura de classes do padrão *Abstract Factory* (figura extraída de GAMMA el al, 2000)

Aplicação no *framework* REMFrame:

No desenvolvimento de *framework*s, o núcleo do sistema deve chamar classes concretas definidas pelo usuário nas diversas aplicações implementadas. Porém o núcleo não conhece estas classes que serão ainda definidas pela instanciação dos *hot spots*. Assim o uso do padrão *Abstract Factory* possibilita ao núcleo do *framework* criar objetos de classes instanciadas pelas aplicações, uma vez que estes objetos são criados através dos métodos declarados nas interfaces das classes fábrica: *ViewFactory, BuilderFactory e AnalysisDataFactory*. Cada uma destas classes define um *hot spot* do *framework*.

Factory Method

O padrão *Factory Method* (GAMMA et al, 2000) define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada.

O padrão *Factory Method* é composto por uma classe *Creator* abstrata que declara o método fábrica o qual retorna um objeto do tipo *Product*. *Product* é uma classe abstrata que define a interface dos objetos criados pelo método fábrica. A classe concreta *Concret Creator* redefine o método fábrica retornando uma instância de um produto concreto (*Concret Product*). (FIGURA 4.6)

No padrão *Factory Method* o cliente não tem que lidar com classes específicas das aplicações, mas somente com a interface de *Product*, podendo trabalhar com quaisquer classes *Concret Product* definidas pelo usuário.

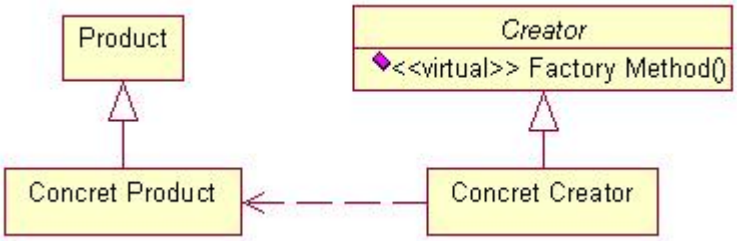

FIGURA 4.6 - Estrutura de classes do padrão *Factory Method*

Aplicação no *framework* REMFrame:

No *framework* REMFrame são métodos fábrica os métodos das classes fábricas citadas para criação de classes de visualização de classes construtoras e classes de definição de dados para transferência. Os métodos fábrica da classe *ViewFactory* retornam referências do tipo *View*. Os métodos fábrica da classe *BuilderFactory* retornam referências do tipo *Builder* e os métodos fábrica da classe *AnalysisDataFactory* retornam referências do tipo *DataManager* e *Domain*. Cada uma das referências retornadas refere-se a uma classe abstrata. Cada aplicação deve definir suas classes concretas, que serão utilizadas pelo núcleo do *framework*.

Builder

Segundo GAMMA et al (2000), o padrão *Builder* separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

Ele é composto pela classe *Director*, que constrói o objeto usando a interface declarada pela classe abstrata *Builder*. A classe *ConcretBuilder* implementa a interface de *Builder* para construir e montar partes de um produto. A classe *Product* representa o objeto complexo construído. (FIGURA 4.7)

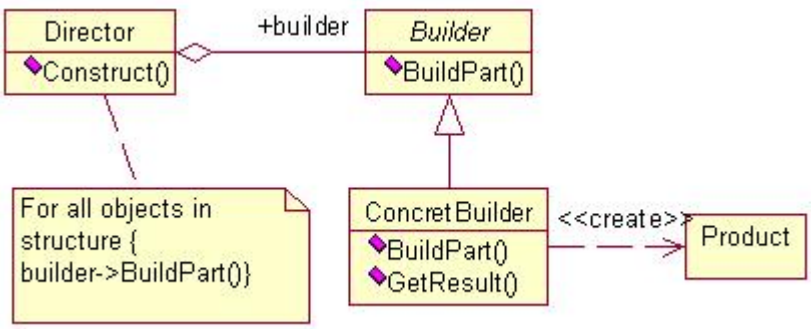

FIGURA 4.7 - Estrutura do padrão *Builder*

Para utilizar o padrão, um cliente cria o objeto *Director* e o configura com o objeto *Builder* desejado. O *Director* notifica o construtor sempre que uma parte do produto deve ser construída e este acrescenta as partes ao produto.

Aplicação no *framework* REMFrame:

O padrão Builder é utilizado para permitir a alteração da construção de diversos objetos manipulados pelo núcleo do *framework* REMFrame. Ele foi implementado pela classe *Builder* do pacote *BuilderClasses* (item 5.1.2). As classes concretas desta hierarquia constroem objetos do tipo *Section, Shape*, *Material*, *Load* e objetos do tipo *Bar*, refinados e inseridos na classe *REMModel*.

Padrões Estruturais

Os padrões estruturais tratam a forma como classes ou objetos podem ser organizados para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Os padrões estruturais de objetos descrevem formas de compor objetos para obter novas funcionalidades.

Whole-Part

O padrão de projeto *Whole-Part* (Todo-Parte) (BUSCHMANN et al, 1996) ajuda na agregação de componentes, que juntos formam uma unidade semântica. Um componente agregado, o Todo, encapsula seus componentes constituintes, as Partes, organiza suas colaborações, e proporciona uma interface comum para suas funcionalidades. O acesso direto às Partes não é permitido. (FIGURA 4.8)

No padrão *Whole-Part* apenas os serviços do Todo são visíveis para clientes externos. O Todo age como um pacote envolvendo suas Partes constituintes,

protegendo-as de acessos não autorizados. Cada Parte é participante de apenas um objeto Todo, sendo criada e destruída dentro do tempo de existência do mesmo.

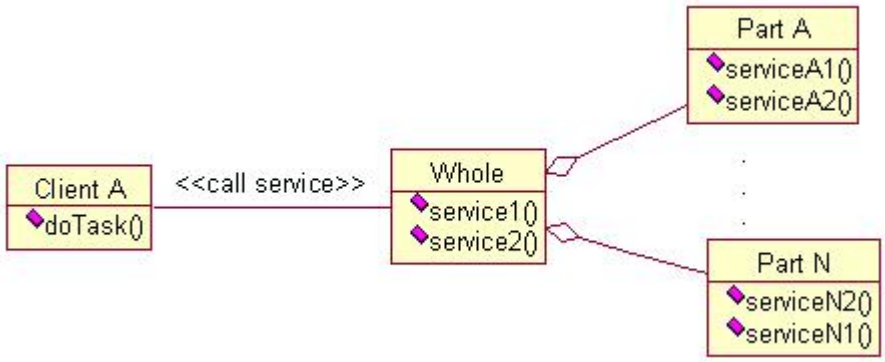

FIGURA 4.8 - Estrutura de classes do padrão *Whole-Part*

Aplicação no *framework* REMFrame:

No desenvolvimento do *framework* o padrão *Whole-Part* foi utilizado nas classes definidas como contêineres (*Project*, *Model*, *LoadCase*), além da classe *ProjectController*, que além do papel de contêiner desempenha a função de fachada para as demais classes do núcleo do *framework*. Estas classes organizam e gerenciam os objetos que representam os modelos da estrutura tratada. A limitação de acesso às partes, definida pelo padrão contribui para um maior controle sobre as intervenções do usuário sobre os dados do modelo. Além disso, este padrão permitiu um maior desacoplamento entre elementos, reduzindo o número de referências entre objetos que passam a obter de seu possuidor (classes contêiner) os dados de outros elementos do modelo (partes) a ele relacionados.

Façade

O padrão *Façade* (GAMMA et al, 2000 e SHALLOWAY, 2004) fornece uma interface unificada para um conjunto de interfaces de um subsistema. Uma classe de fachada define uma interface de mais alto nível, tornando o uso do subsistema mais

fácil. Ao promover um acoplamento fraco entre o subsistema e seus clientes, este padrão possibilita uma substituição mais fácil dos subsistemas utilizados.

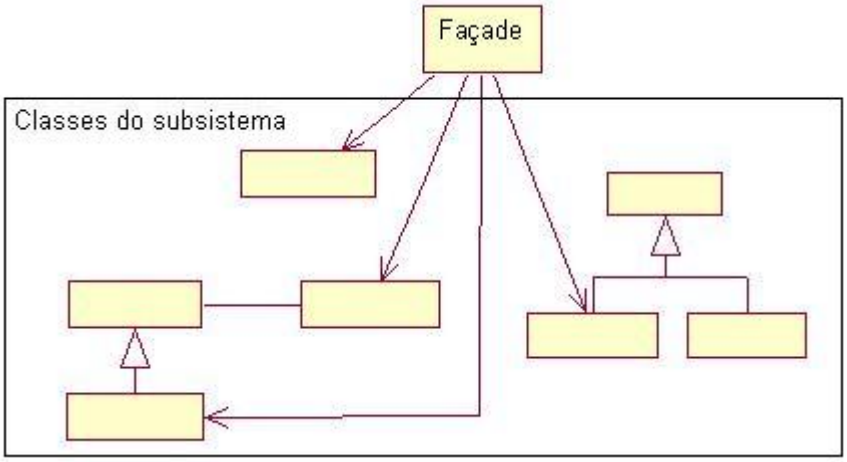

FIGURA 4.9 - Estrutura do Padrão *Façade*

O padrão *Façade* é implementado através de uma classe fachada que reúne as classes que implementam a funcionalidade de um subsistema sem ocultá-las completamente. Por conhecer quais são as classes de um subsistema responsáveis pelo atendimento de uma solicitação, a classe *Façade* delega as solicitações de clientes a objetos apropriados do subsistema (FIGURA 4.9).

Aplicação no *framework* REMFrame:

A classe *ProjectController* do REMFrame desempenha a função da classe fachada desse padrão. Ela recebe as solicitações dos usuários através das aplicações, faz um tratamento inicial dos dados quando necessário, identifica as classes do sistema que respondem à solicitação e repassa as mesmas para as devidas classes. Apesar de ser uma classe concreta, a definição de classes derivadas desta classe nas aplicações desenvolvidas permite deixar para as mesmas a implementação de algumas definições necessárias à configuração geral do sistema, principalmente no que diz respeito ao

tratamento de dados antes da transferência de solicitações às classes do núcleo do *framework* ou para os servidores internos.

Bridge

Segundo GAMMA et al (2000) o padrão *Bridge* desacopla uma abstração de sua implementação, de modo que as duas possam variar independentemente. Se uma abstração pode ser representada por uma hierarquia de classes e uma implementação pode ser representada por outra hierarquia, ao relacionar as duas hierarquias é possível evitar um vínculo permanente entre uma implementação e sua representação. Este relacionamento é obtido através de composição onde uma classe base A possui uma referência para a outra classe base B. Através desta referência qualquer classe da hierarquia A pode receber um objeto de qualquer classe da hierarquia B. A estrutura deste padrão é apresentada na FIGURA 4.10.

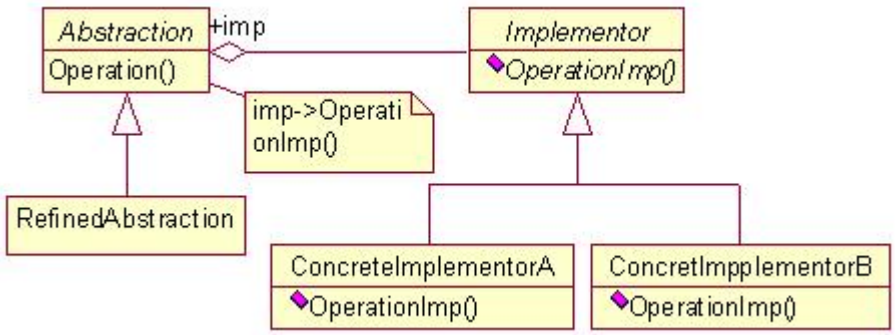

FIGURA 4.10 - Estrutura de classes do padrão Bridge

Aplicação no *framework* REMFrame:

O padrão Bridge é aplicado no *framework* REMFrame para a implementação das diferentes seções geométricas simples (hierarquia da classe *Shape*) e o cálculo de suas diversas propriedades geométricas (hierarquia da classe *GeomPropHandler*)

Adapter

O padrão *adapter* (GAMMA et al, 2000 e SHALLOWAY, 2004) permite converter a interface de uma classe em outra utilizada pelos clientes. Através deste padrão um cliente chama um método de uma classe adaptador e esta repassa a chamada para outra, que executa a solicitação. Assim é possível utilizar uma classe cuja interface não é conhecida pelo cliente sem necessitar alterá-la. A FIGURA 4.11 apresenta a estrutura para o padrão *Adapter* utilizada pelo framework REMFrame.

Aplicação no *framework* REMFrame:

No REMFrame o padrão *Adapter* é utilizado para a definição das classes de visualização. Este padrão permite que a funcionalidade gráfica da plataforma seja utilizada pelas classes derivadas de *View*. O método *Draw* destas classes aciona os métodos de desenho das classes da plataforma gráfica que implementam a visualização dos elementos.

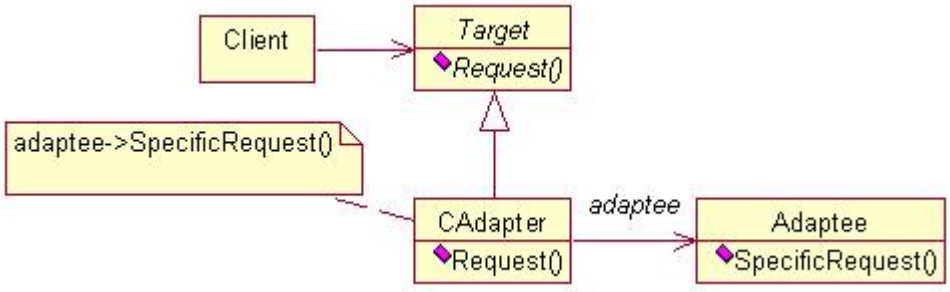

FIGURA 4.11 - Estrutura do padrão *Adapter*

Padrões Comportamentais

Padrões comportamentais encapsulam variações, definindo um objeto responsável por encapsular o aspecto de um programa que muda freqüentemente. Estes padrões descrevem aspectos mutáveis de programas e a maioria deles tem dois tipos de

objetos: os novos, que encapsulam o aspecto e os existentes, que utilizam os novos objetos.

Observer

O padrão *Observer* (GAMMA et al, 2000) define uma dependência entre objetos de modo que, quando um objeto muda seu estado, todos os aqueles que dele dependem são automaticamente notificados e atualizados, mantendo a consistência entre os objetos relacionados.

O padrão é formado por uma classe abstrata *Subject* que fornece uma interface para associar e desassociar objetos *observers*. Um objeto *subject* pode ter um número qualquer de observadores. A classe abstrata *Observer* define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um *Subject* (FIGURA 4.12). A classe *ConcreteSubject* armazena estados de interesse para objetos *ConcreteObserver* e envia uma notificação para seus observadores quando seu estado muda. A classe *ConcreteObserver* mantém uma referência para um objeto *ConcreteSubject*, armazena estados que deveriam permanecer consistentes com os do *Subject* e implementa a interface de atualizações de *Observer*.

Um *observer* pode observar mais de um *subject* e neste caso ele deve saber qual *subject* está disparando uma notificação. Para isso o *subject* pode passar a si mesmo como parâmetro para a função *Update*, permitindo ao observer saber qual *subject* deve examinar.

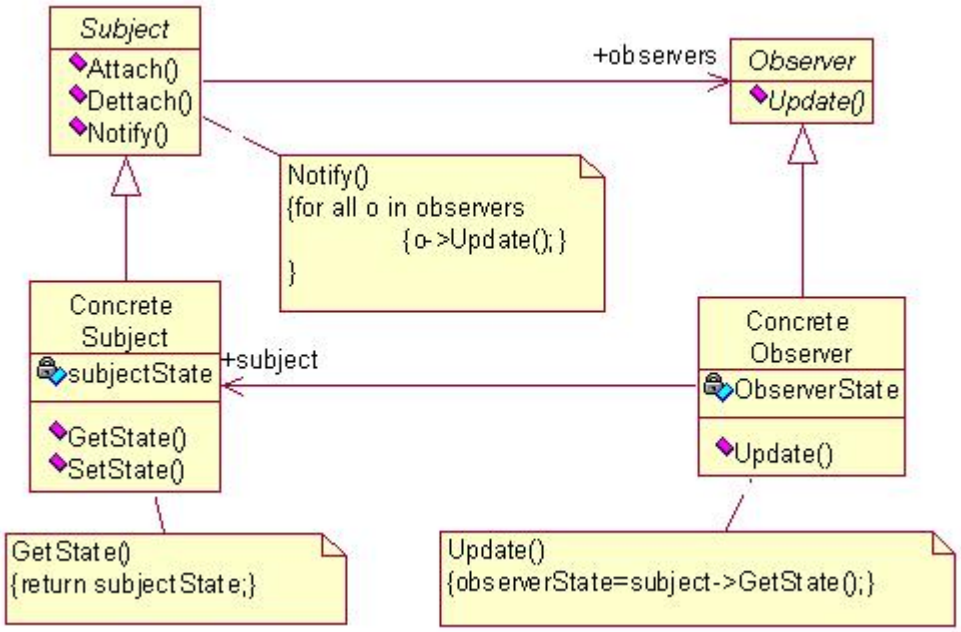

FIGURA 4.12 - Estrutura de classes do padrão *Observer*

Aplicação no *framework* REMFrame:

No REMFrame este padrão é utilizado para implementar os relacionamentos entre as classes de dados, derivadas de *Subject*, e as classes de visualização, derivadas de *View*. A classe *View* por sua vez é derivada da classe *Observer*. Este padrão foi implementado no *framework* através da definição das classes *Subject* e *Observer* organizadas no componente *ClassesMVC*.

Strategy

Segundo GAMMA et al (2000), o padrão *Strategy* "define famílias de algoritmos, encapsulando cada família e tornando-as intercambiáveis, permitindo que o algoritmo varie independentemente dos clientes que o utilizam." Ele é utilizado quando a diferença entre muitas classes relacionadas concentra-se em seu comportamento. Para implementar o padrão uma classe *Context* possui uma referência para uma classe

abstrata (*Strategy*) que declara uma interface para um comportamento. Para variar o comportamento ele deve ser definido nas classes *ConcretStrategy*. (FIGURA 4.13).

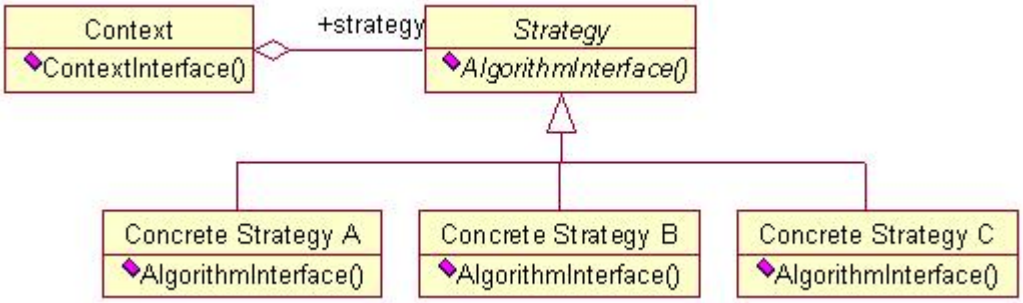

FIGURA 4.13 - Estrutura de classes do padrão *Strategy*

Strategy e *Context* interagem para implementar o algoritmo escolhido. Os clientes criam e passam um objeto *ConcreteStrategy* para o contexto, interagindo em seguida exclusivamente com este, que repassa solicitações dos clientes para a sua estratégia.

Aplicação no *framework* REMFrame

Este padrão foi utilizado no *framework* para permitir o tratamento de dados de forma diferenciada para transferência entre o *frozen spot* e cada sistema externo utilizado. As classes *DataManager e Builder* desempenham o papel da classe *Strategy,* enquanto a classe *ProjectController* representa a classe *Context.* As aplicações são responsáveis por definir e instanciar as classes derivadas de *DataManager* e de *Builder*, configurando a classe *ProjectController* com as mesmas.

Este padrão também foi aplicado para a implementação do cálculo das propriedades geométricas das seções transversais através da hierarquia da classe *GeomPropHandler*.

O padrão *Strategy* também foi adotado para a implementação dos métodos de relacionamento ou de associação entre os elementos de ligação (*Conection*) e os objetos do tipo *Bar*. Estes métodos são declarados na interface da classe abstrata *ConectionRules* e devem ser implementados nas aplicações que façam o detalhamento geométrico das ligações.

O padrão *Strategy* foi utilizado também na transferência de dados do REMFrame, onde a classe base *Component* define em sua interface os métodos de *FillIn* e *FillOut* para preenchimento e persistência de objetos complexos, repsectivamente. Estes métodos ao serem implementados em cada classe derivada, permitem a gravação e leitura de todos os objetos de dados em documentos externos.

Template Method

O padrão *Template Method* (GAMMA et al, 2000) define o esqueleto de um algoritmo em uma operação, permitindo que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

Este padrão implementa as partes invariantes de um algoritmo deixando para as subclasses a implementação do comportamento que pode variar.

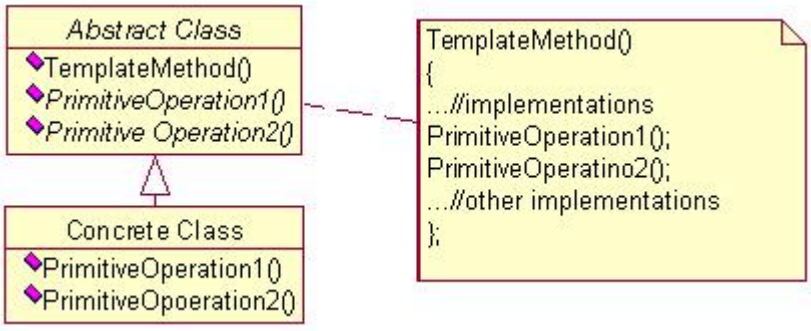

FIGURA 4.14 - Estrutura de classes do padrão *Template Method*

O padrão é formado por uma classe abstrata que define operações primitivas abstratas, as quais as subclasses concretas definem para implementar os passos de um algoritmo(FIGURA 4.14).

Aplicação no *framework* REMFrame

Cada método da classe *ProjectController* que implementa um caso de uso do *framework* foi estruturado segundo este padrão. Os fluxos foram desmembrados em operações que são acionadas por um método geral. Algumas destas operações foram definidas como métodos virtuais, permitindo sua redefinição por classes derivadas e implementadas pelas aplicações, alterando parcialmente a funcionalidade do caso de uso como nos casos de uso *NewSection*, *NewMaterial* e *AddLoad*, onde a forma de definição dos objetos pode ser redefinida pela aplicação. As classes contêiner também apresentam métodos estruturados segundo este padrão. No caso de uso *InsertBar* as operações que definem a permissão de interseção entre elementos ou a ordem de numeração de elementos podem ser definidas pelas implementações, alterando os critérios de inserção de elementos.

4.3 Os instrumentos utilizados

Para a implementação do *framework* foram utilizados: a linguagem de programação C++, os paradigmas da POO, padrões de arquitetura e de projeto catalogados por GAMMA et al (2000) e BUSCHMANN et al (1996) e descritos nos item 4.2.

A linguagem C++ foi escolhida por ser a linguagem de desenvolvimento de algumas plataformas gráficas de grande veiculação comercial, e de arquitetura aberta, que possibilitam o desenvolvimento de aplicações personalizadas utilizando a

funcionalidade já implementada em sua estrutura na implementação de novos sistemas clientes.

Os sistemas especializados desenvolvidos a partir do *framework* foram implementados utilizando o ambiente de programação ObjectARX® [22](AutoCAD® Runtime Extension). O ObjectARX (AUTODESK, 2006) é o responsável por tornar classes e funções utilizadas pelo AutoCAD disponíveis para implementações específicas.

Para as etapas de análise e desenho do sistema foi empregada a *Unified Modeling Language* (QUATRANI, 1998 e FURLAN, 1998).

O *framework* foi desenvolvido permitindo sua integração a diferentes plataformas gráficas e a outros *frameworks*. A interface de usuário dos primeiros sistemas clientes implementados foi desenvolvida com o uso do *framework* MFC (*Microsoft Foundation Classes*) e da plataforma gráfica AutoCAD 2006. Optou-se por utilizar esta plataforma gráfica para as primeiras implementações objetivando-se uma análise comparativa com os demais sistemas gráficos analisados e desenvolvidos como aplicações sobre outras versões da mesma plataforma.

O Microsoft Development Environment 2002, Version 7.0 foi adotado como ambiente de desenvolvimento tendo o Microsoft Visual C++.NET como compilador (MICROSOFT, 2004). Estas escolhas foram condicionadas pela escolha de utilização inicial do ObjectARX para a implementação das visualizações do modelo. A utilização

---

[22] ObjectARX® é o ambiente de programação do AutoCAD® que inclui bibliotecas em C++ que são blocos de construção utilizados para desenvolver aplicações para o AutoCAD, classes extendidas, protocolos e criar novos commandos que operam do mesmo modo que os comandos internos do AutoCAD

deste ambiente de programação é predefinida pela Autodesk para o desenvolvimento de aplicativos para o AutoCAD versões 2004 a 2006. Assim como o Visual C++, esse ambiente integrado de projeto permite uma programação mais rápida e eficiente devido as suas ferramentas internas de programação (CULLENS *et al*, 1997 e DEITEL, 1997). Neste ambiente é possível utilizar a ferramenta de programação ObjectARX Wizards, que auxilia a criação dos aplicativos com derivações por herança das classes das bibliotecas disponíveis, acesso às estruturas de banco de dados e ao sistema gráfico do AutoCAD, além da criação de novos comandos, que operam do mesmo modo que os comandos nativos do AutoCAD.

4.4 Técnicas de Pesquisa e os procedimentos adotados

A presente pesquisa foi desenvolvida em três etapas principais: pesquisa bibliográfica, pesquisa documental e pesquisa de laboratório. Durante as pesquisas bibliográfica e documental foram avaliados alguns trabalhos desenvolvidos pelo grupo CADTEC, além de sistemas, ambientes e um *framework* orientado a objetos para análise via MEF (Capítulo 3). Também foram pesquisados nestas etapas os conceitos e teorias referentes a *frameworks* e a padrões (itens 4.1 e 4.2). Após estas duas etapas iniciou-se o desenvolvimento do *framework*. Esta etapa de desenvolvimento foi baseada tanto nas fases definidas por MARKIEWICZ e LUCENA (2001) (FIGURA 4.2), quanto na Linguagem de Padrões para o Desenvolvimento de Frameworks Orientados a Objetos apresentada por ROBERTS e JOHNSON (1996).

Uma descrição dos procedimentos e decisões adotadas nas etapas de análise de domínio, projeto do *framework* e implementação será apresentada a seguir.

Na etapa de análise de domínio foram desenvolvidas as seguintes atividades:

- Identificação do domínio, de sua extensão e de seus requisitos. A pesquisa bibliográfica realizada foi fator preponderante para a identificação do domínio.

- Levantamento das etapas gerais do processo produtivo de estruturas reticuladas.

- Levantamento de pré-requisitos para projeto de estruturas reticuladas.

- Pesquisa de campo com visitas a empresas de pré-fabricados, para conhecimento do processo de produção destas estruturas, seu nível de automação e integração entre suas diversas etapas.

- Estudo de programas de modelagem e de dimensionamento de estruturas reticuladas com foco em funcionalidades gerais, limitações, interface com o usuário, entradas de dados e saídas dos resultados. Nesta etapa foram analisados os sistemas Premoldar, TQS, Mudados, INSANE, além de outros sistemas acadêmicos desenvolvidos e utilizados pelo Departamento de Engenharia de Estruturas da UFMG.

- Estudo de sistemas CAD desenvolvidos para a modelagem de estruturas reticuladas com identificação de similaridades entre os sistemas e de particularidades de cada um (pesquisa bibliográfica).

- Avaliação da viabilidade do desenvolvimento de um *framework* para a geração de sistemas integradores das etapas do processo produtivo de estruturas reticuladas (LEITE e FRANCO, 2006a) (LEITE e FRANCO, 2006b).

Na etapa de desenho ou projeto do *framework* foram realizados estudos visando definir a estrutura e a funcionalidade do sistema (itens 5.1 e 5.2). Posteriormente foram lançados o desenho do sistema, com definição de componentes, classes e fluxos de dados. Nesta fase foram desenvolvidas as seguintes atividades:

- Estudo de padrões de arquitetura e de projeto.

- Escolha dos padrões de arquitetura e de projeto aplicáveis ao *framework* (LEITE e FRANCO, 2005)

- Identificação de módulos, de classes e do relacionamento entre as mesmas, necessários à modelagem e ao detalhamento de estruturas reticuladas (LEITE e FRANCO, 2004) e(LEITE e FRANCO, 2006b).

- Identificação de *frozen spots* e *hot spots* para o *framework* (LEITE e FRANCO, 2004) e(LEITE e FRANCO, 2006b).

- Definição das interfaces com os programas de análise e de dimensionamento e das interfaces com o usuário.

- Definição da forma de manipulação dos dados através de arquivos de interface.

Nesta etapa, definiu-se pelo uso dos padrões de arquitetura MVC e *Microkernel* para o desenvolvimento do *framework.* A adoção destes dois padrões resultou em uma estrutura de classes onde adotou-se uma classificação comportamental para as classes (modelo, visualização ou controle) além de uma organização modular ou por componentes funcionais (servidores internos, servidores externos, núcleo, adaptador e clientes).

A etapa de projeto foi desenvolvida baseada nos passos fundamentais apresentados por BUSCCHMANN *et al* (1996) para a criação de uma aplicação utilizando-se o padrão MVC, porém a ordem das atividades e os procedimentos adotados diferem daqueles propostos pelo referido autor uma vez que o padrão MVC foi aplicado com algumas alterações, como descrito a seguir:

1. Inicialmente foi definido e implementado o mecanismo de propagação de mudanças. Utilizando o padrão *Observer*, foi definida uma classe base *Subject* que armazena ponteiros para a classe *Observer* através de uma lista encadeada. Foram definidas duas classes derivadas da classe *Observer*: *View* e *Controller*. Foram definidos e implementados os métodos *attach* e *detach* para permitir que *views* e *controllers* sejam conectados ou desconectados a

um objeto da classe *Subject*. Nesta classe foi declarado o método *notify*. Este método aciona o método *update*, declarado na classe *Observer,* para cada observador anexado a um elemento de modelo (*Subject*). Todos os procedimentos do modelo, que alteram seu estado são responsáveis por acionar o método *notify* após a realização das mudanças.

2. A classe base e abstrata *View* foi definida, sendo especificado e implementado um procedimento *update* para refletir as mudanças informadas pela classe *Subject*. Este método recebe um ponteiro para o elemento de modelo alterado e a partir da validação deste ponteiro é acionado o método *draw* ou o método *erased* para a visualização. Estes procedimentos foram definidos através do padrão *Template Method,* devendo ser implementados em cada classe concreta derivada da classe *View*. A classe *View* implementa ainda os procedimentos de inscrição no mecanismo de propagação de mudanças ao acionar, através de sua construtora, o método *attach* de seu *Subject*, e de retirada da inscrição ao acionar o método *detach* de seu *Subject* pela sua destrutora.

3. Em seguida os elementos de modelo, de visualização e de controle foram classificados quanto a sua função específica dentro do domínio. Esta classificação buscou identificar um núcleo mínimo de dados e de procedimentos, agrupando em componentes independentes operações e dados que seriam acionados por este núcleo mínimo. Foram então definidos os componentes: componente para manipulações matemáticas (*MathClasses*), componente de propriedades físicas (*MaterialClasses*), componente de definição geométrica (*SectionClasses*) e componente de transferência de dados (*FillerClasses*). Foram definidas classes específicas para armazenar e organizar as classes de modelo (*Project*, *Model* e *LoadCase*). Estas classes desempenham o papel tanto de contêiner ao armazenar dados, quanto de controle ao organizar o fluxo de dados e repassar as solicitações do usuário para as classes que armazena (padrão *Whole-Part*). Estas classes foram identificadas como classes de banco de dados, pois armazenam os elementos do modelo em dicionários, onde os

elementos são identificados por identificadores, de maneira semelhante a tabelas em um banco de dados.

4. As classes de modelo foram definidas para encapsular os dados e a funcionalidade do núcleo, provendo funções para acessar os dados do mesmo. A classe de controle *ProjectController* foi definida como uma interface entre as classes de banco de dados e as aplicações evitando que funcionalidades do modelo fossem expostas diretamente ao usuário. Esta intermediação foi criada uma vez que a inserção de dados no modelo depende de uma verificação constante de consistência dos dados e de validações no modelo. A classe de controle é então a única que recebe e interpreta solicitações de usuário, repassando os mesmos para as classes de banco de dados e estas para as classes de modelo.

5. A classe de controle seleciona as classes de banco de dados às quais a informação é repassada. Estas realizam as validações de informações e verificações de consistência antes de inserir, retirar ou alterar um dado do modelo. Apesar de não ter sido desenvolvido um controlador para cada visualização do modelo, como proposto pelo padrão MVC, a cada inserção ou retirada de elementos no modelo, uma das classes de banco de dados, faz a vinculação, ou desvinculação, entre uma visualização e o elemento de modelo inserido ou retirado.

6. Foram selecionados os padrões de projeto apropriados para modelar cada tipo de flexibilização requerida pelo sistema. Definidos estes padrões foram projetados e codificados os *hot spots* do sistema através da definição de várias classes abstratas. Entre estas classes estão as classes para implementação de visualização (*View*), leitura e gravação de dados (*Filler*), formatação de dados (*DataManager*), configuração de dados de ambiente para a análise, para o dimensionamento e para o detalhamento da estrutura reticular (*Domain*).

7. Foi utilizado o padrão *Abstract Factory* para permitir que a implementação de cada visualização concreta fosse realizada apenas nos sistemas clientes.

> Através deste padrão o *framework* é capaz de vincular uma visualização a cada elemento do modelo sem necessariamente conhecer a visualização concreta utilizada permanecendo independente de *views* específicas. A especificação e a implementação dos procedimentos *draw* e *erased* para cada visualização criada podem ser efetuadas nos sistemas clientes, permitindo, então a variação da plataforma gráfica utilizada. Esta variação também é garantida pela independência do sistema com relação às manipulações geométricas dos dados do modelo. O componente de classes matemáticas é o único responsável por tais manipulações, ficando a plataforma gráfica responsável somente pela representação, através de entidades gráficas, dos dados informados pelos elementos de modelo representados.

Na etapa de implementação foram desenvolvidos alguns aplicativos clientes. A primeira aplicação desenvolvida foi um modelador implementado como um sistema que utiliza a plataforma AutoCAD para as representações gráficas. Este sistema disponibiliza comandos para entrada de dados e implementa as representações gráficas do modelo, funcionando tanto como pré-processador quanto como pós-processador. Foram implementadas classes para a representação do modelo geométrico unifilar, representação do modelo geométrico sólido pela adição de uma seção transversal ao longo dos elementos do modelo, visualizações dos dados oriundos de condições de contorno como representações de carregamentos e de restrições nodais, visualização da estrutura deformada e de gráficos dos esforços solicitantes, aos quais a estrutura está submetida.

Dois outros sistemas desenvolvidos foram os sistemas PREMOLD e STEELMOLD para lançamento automatizado de elementos pré-fabricados de concreto e perfis usinados respectivamente. Estes sistemas foram desenvolvidos tendo como base o modelador gráfico descrito anteriormente, implementando interfaces de usuário para a entrada de dados, para a inserção de materiais e inserção de seções transversais através da automação de leitura de tabelas de perfis comerciais. Estas implementações permitiram uma avaliação comparativa entre os novos sistemas gerados, suas características e limitações em relação aos sistemas Modelador 3D, TowerCAD e PREMOLD (original). Estas avaliações comparativas forneceram parâmetros para uma

melhor proposição dos procedimentos e implementações necessárias para permitir ao *framework* armazenar e gerenciar dados de detalhamentos de ligações.

Posteriormente foram implementados dois sistemas cliente que fazem a integração entre um sistema CAD e um sistema CAE. O modelador gráfico originalmente criado foi utilizado como pré e pós-processador para dois sistemas de análise via método dos elementos finitos a saber o NLG (GRECO, 2006) e o PLANLEP (LAVALL[23] *apud* SILVA R., 2004). O sistema NLG faz a análise não linear geométrica de estruturas bidimensionais e tridimensionais, utilizando elementos de treliça. O sistema PLANLEP executa a análise avançada de pórticos planos de aço. O desenvolvimento destes sistemas de integração permitiu a avaliação dos *hot spots* de interface de arquivos, a validação da estrutura de dados proposta e do sistema de geração da malhas de elementos lineares implementada pelo *framework*.

Estas implementações permitiram a avaliação do *framework* quanto a sua aplicabilidade no desenvolvimento de sistemas para o processo produtivo de estruturas reticuladas. Foram avaliados os procedimentos necessários para o desenvolvimento destes sistemas e as soluções obtidas, comparando os resultados obtidos aos esforços desprendidos para o desenvolvimento dos demais sistemas analisados ao longo desta pesquisa e que impulsionaram seu desenvolvimento.

---

[23] Lavall, A. C. C., 1996. Uma Formulação Teórica Consistente para a Análise Não Linear de Pórticos Planos pelo Método dos Elementos Finitos Considerando Barras com Imperfeições Iniciais e Tensões Residuais nas Seções Transversais. Tese de Doutorado. Escola de Engenharia de São Carlos, USP.

5

UM FRAMEWORK PARA AUTOMAÇÃO/INTEGRAÇÃO DO PROCESSO DE DESENVOLVIMENTO DE PROJETO DE ESTRUTURAS RETICULADAS TRIDIMENSIONAIS

Neste capítulo é apresentada a arquitetura do REMFrame organizada em componentes e pacotes segundo uma adaptação do padrão *Microkernel*. Em seguida são listados os *hot spots* definidos e os principais casos de uso implementados.

5.1 Arquitetura e Desenho

Conforme citado no item 4.2.4, o *framework* foi estruturado segundo dois padrões de arquitetura: o *Model View Controller* (MVC) e *Microkernel.* Ambos os padrões apresentam propostas de estruturação que visam garantir uma flexibilização do sistema: o MVC permite a flexibilização da interação com o usuário enquanto o *Microkernel* flexibiliza a adaptação e extensão do sistema.

Segundo a arquitetura MVC o sistema foi organizado em três tipos básicos de classes: classes responsáveis por armazenar os dados do sistema estrutural, classes responsáveis por fornecer uma representação gráfica dos dados e classes responsáveis por controlar fluxos de dados e de solicitações de alteração de estado. Estas classes foram organizadas, segundo o padrão *Microkernel*, em seis componentes estruturados como bibliotecas de vínculo dinâmico (FIGURA 5.1): *ClassesMVC, FillerClasses, MathClasses, MaterialClasses, SectionClasses, Kernel.* Estes componentes serão detalhados nos itens 5.1.1 e 5.1.2.

Os componentes *ClassesMVC*, *FillerClasses, MaterialClasses, SectionClasses* e *MathClasses* são componentes que desempenham o papel de servidores internos segundo a arquitetura *Microkernel* e organizam classes que oferecem funcionalidades gerais ao sistema, sendo acionadas pelo núcleo do *framework* sempre que este necessita realizar a manipulação de dados geométricos e a leitura ou a gravação de dados.

O componente *Kernel* representa o componente *microkernel* do padrão de mesmo nome.

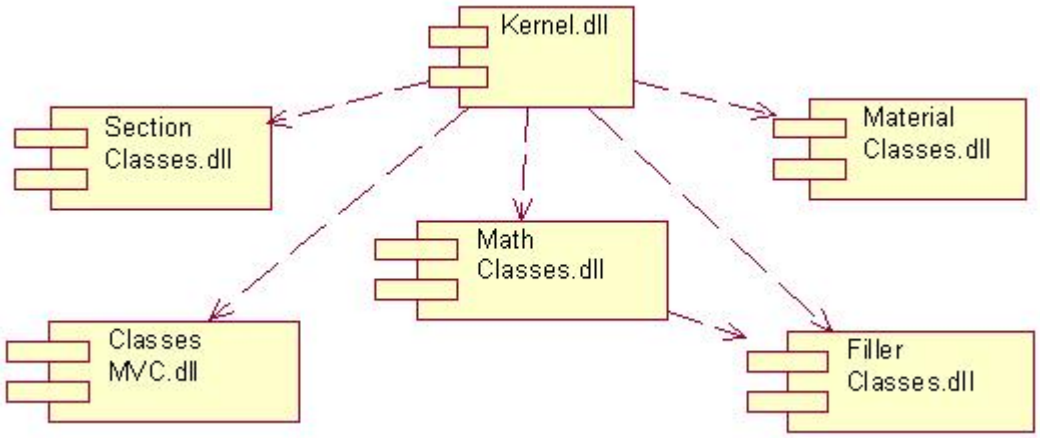

FIGURA 5.1 – Componentes do *framework* REMFrame

5.1.1 Componentes de base ou servidores internos

Componente *ClassesMVC*

No componente *ClassesMVC* foram inseridas as classes que modelam o padrão de arquitetura MVC e implementam o mecanismo de propagação de mudanças através do padrão de projeto *Observer*. Neste componente foi inserida também a classe base *Component* que define métodos para a persistência dos dados (*FillIn* e *FillOut*) e métodos de identificação de uma classe (*isA*) e de um objeto dessa classe, através de seu identificador (*Number*) (FIGURA 5.2). Os métodos *FillIn* e *FillOut* foram criados segundo o padrão *Strategy* e devem ser implementados em todas as classes derivadas para definir a persistência dos dados de objetos complexos. Através destes métodos virtuais é possível persistir e recompor objetos concretos não conhecidos pelo núcleo do *framework*. Este componente reúne, então, as classes base para praticamente todas as demais classes do *framework*: *Subject, View, Controller e Component*.

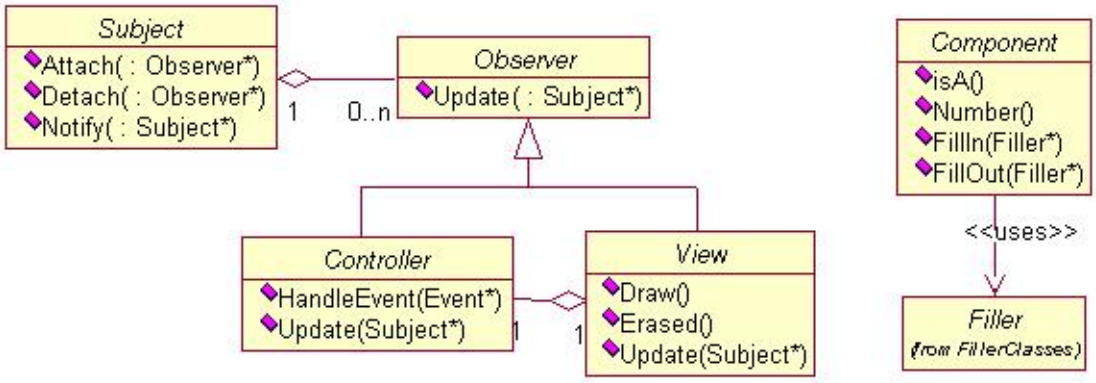

FIGURA 5.2 – Classes do componente *Classes MVC*

Componente *FillerClasses*

No *framework* as classes de visualização, tratam da interface com o usuário e com os programas externos, gerenciando a saída gráfica e textual, por intermédio dos *hot spots* representados pelas classes *View*, *Filler* e *DataManager*. Esta última será descrita no pacote *DataManager Classes* do componente *Kernel*.

A classe *Filler* é uma classe abstrata que define uma interface para as classes de transferência de dados, declarando métodos que efetuam a leitura ou a gravação de tipos básicos da linguagem (números, caracteres e conjunto de caracteres) e de objetos complexos definidos no componente *Math Classes*, como objetos da classe *Values*. Esta classe, juntamente com a classe base *DataManager,* permite a variabilidade da transferência de dados, adequando a mesma à formatação requerida pelos sistemas integrados pelo *framework* ou permitindo a definição de diferentes interfaces de arquivos. Também a interface de usuário pode ser modelada por esta classe para permitir diferentes formatos de entrada de dados nos sistemas clientes desenvolvidos, utilizando métodos próprios de um determinado ambiente ou sistema (por exemplo DOS, MFC, ou AutoCAD).

O componente *Filler Classes* é formado pela classe abstrata *Filler*. Foi também definida, nesta versão do *framework*, uma classe concreta, *FileFiller,* que implementa as

funções de leitura e gravação de dados para arquivos ASCII. As classes do componente *Filler Classes* são mostradas na FIGURA 5.3.

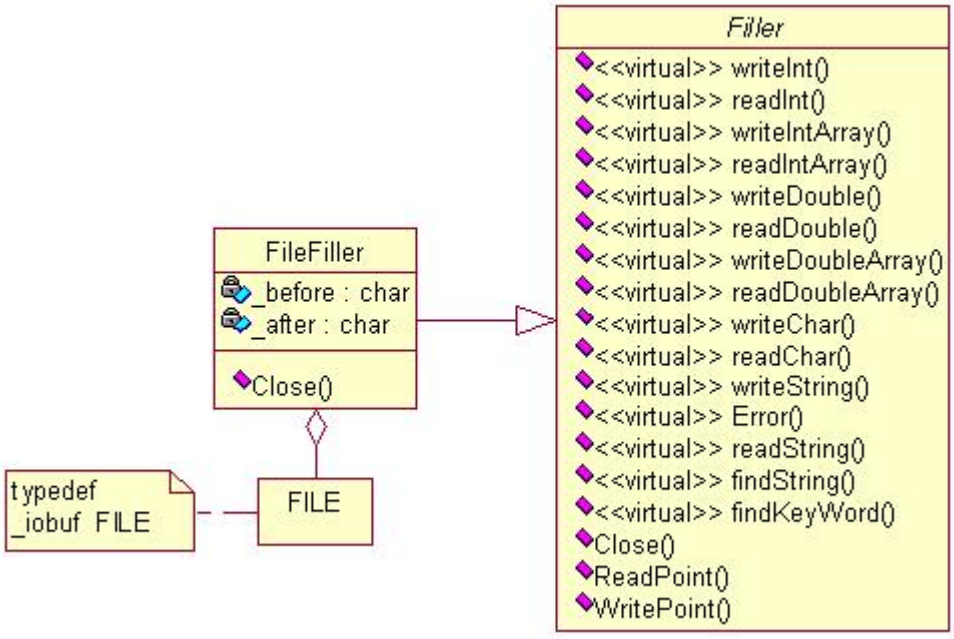

FIGURA 5.3 – Classes do componente *Filler Classes*

Componente *Math Classes*

O componente *Math* Classes foi desenhado para representar grandezas matemáticas, além de desempenhar o cálculo e o tratamento de dados geométricos. Este componente implementa manipulações geométricas básicas, permitindo cálcular e manipular as coordenadas cartesianas de elementos, as interseções entre elementos e as validações de dados. Estas funcionalidades são disponibilizadas pelo núcleo do *framework* aos sistemas clientes, que podem trabalhá-las de forma independente em relação à plataforma gráfica escolhida. Neste componente estão agrupadas as classes *Point*, *Line*, *Axis*, *CoordinateSystem*, *Plane*, *Matrix*, *Interpolation e Values* (FIGURA 5.4). Estas classes, segundo a arquitetura MVC são classificadas como classes de modelo segundo BURBECK (2003), pois gerenciam o comportamento dos dados, respondendo a solicitações repassadas, geralmente, através de objetos controladores.

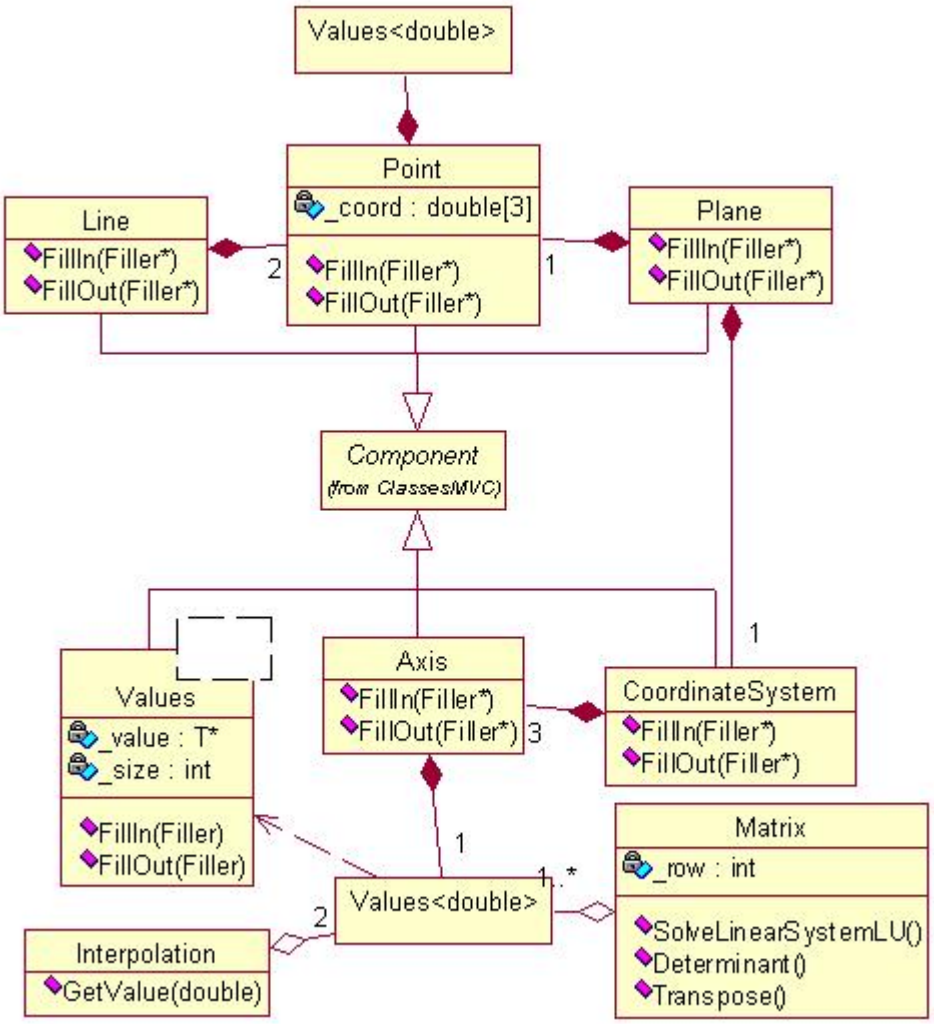

FIGURA 5.4 – Classes do componente *Math Classes*

As classes *Line*, *Point*, *Plane*, *Axis* e *CoordinateSystem* representam entidades geométricas. A classe *Line* representa um segmento de reta no espaço tridimensional, definida por um objeto do tipo *Point* em cada uma de suas extremidades. A classe *Point* representa um ponto do espaço Euclidiano, definidos por três coordenadas. A classe *Axis* representa um vetor direcional. A classe *CoordinateSystem* representa um sistema de coordenadas cartesianas, formado por três eixos (*Axis*) perpendiculares entre si. A classe *Plane* representa um plano no espaço tridimensional sendo definido por um ponto (*Point*) e um vetor normal ao plano (*Axis*).

As classes *Matrix* e *Values* representam conjuntos de valores relacionados. A classe *Values* implementa um conjunto de dados parametrizados e a classe *Matrix* representa uma matriz de valores numéricos e implementa os métodos *SolveLinearSystemLU, Transpose* e *Determinant*, utilizados por outros métodos que realizam a manipulação dos dados geométricos como, por exemplo, na verificação de pontos de interseção entre elementos lineares no espaço tridimensional.

A classe *Interpolation* foi declarada para permitir o tratamento de grandezas de distribuição não linear através de funções polinomiais, como, por exemplo, cargas distribuídas não lineares.

Componente *Section Classes*

O componente *Section Classes* reúne a classe *Section* e as classes base *Shape* e *GeomPropHandler* (FIGURA 5.5). Foram definidas três classes derivadas de *Shape*: *Circle*, *CircularSection* e *Polygon*. Através destas três classes é possível representar seções circulares, seções poligonais e seções formadas por setores circulares. Um objeto do tipo *Section* é formados por um conjunto de *Shapes*, sendo capaz de representar seções transversais compostas por diferentes formas geométricas. A classe *GeomPropHandler* define a interface de métodos para cálculo das propriedades geométricas de seções transversais planas. Através dos padrões *Strategy* e *Bridge* as classes *Section*, *Shape* e *GeomPropHandler*, permitem a representação de diferentes seções transversais e a obtenção de suas propriedades geométricas. Para a obtenção das propriedades de uma seção composta a classe *Section* solicita a cada objeto do tipo S*hape* que retorne suas propriedades, efetuando o somatório das propriedades retornadas por cada componente da seção e retornando o resultado final obtido. Um objeto do tipo Shape possui uma referência para objetos do tipo *GeomPropHandler,* repassando a estes a obtenção das propriedades requeridas pelo objeto *Section.* Assim. o cálculo das propriedades geométricas foi implementado pela utilização do padrão *Strategy* (item 4.2.5) onde classes derivadas da classe *GeomPropHandler* implementam o cálculo das propriedades para as diferentes formas geométricas (*Shapes*).

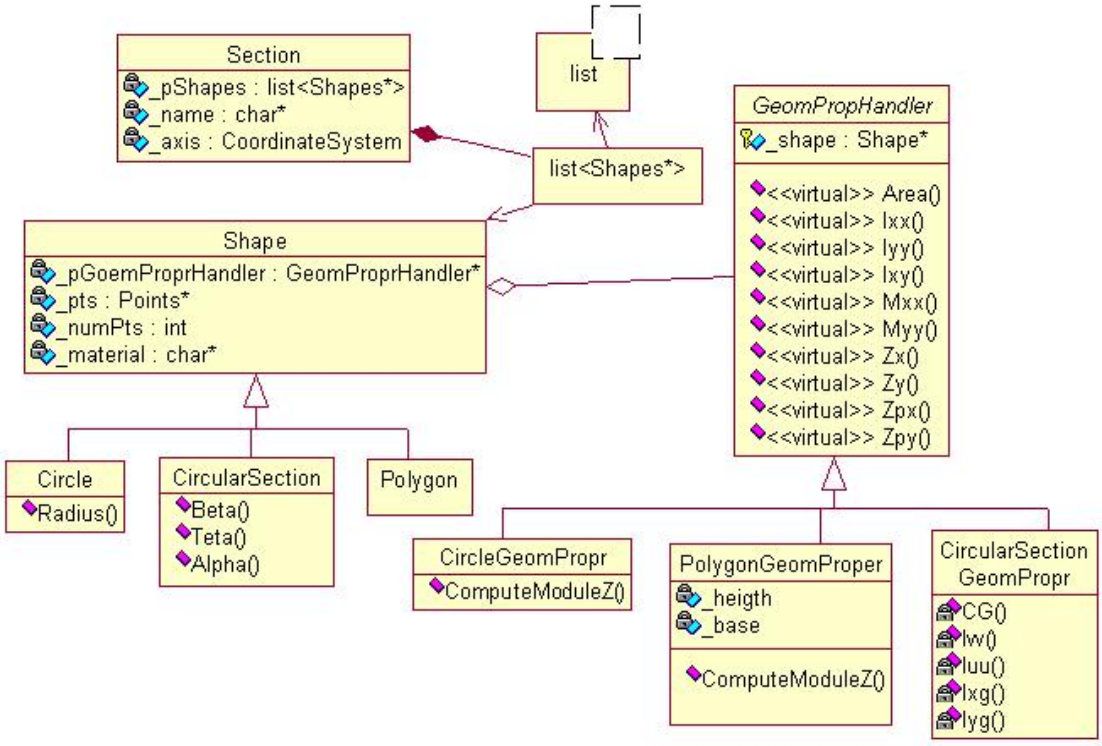

FIGURA 5.5 – Classes do componente *Section Classes*

Componente *Material Classes*

O componente *Material Classes* engloba a hierarquia da classe *Material* responsáveis por armazenar e gerenciar os dados relativos aos diversos tipos de materiais presentes nos elementos da estrutura. A classe *PropertyMap* (FIGURA 5.6) é responsável por armazenar os dados relativos às propriedades dos materiais. Esta classe funciona como um dicionário de propriedades, permitindo armazenar diferentes valores, que podem ser aplicados em diferentes direções, para cada uma das propriedades. Na hierarquia da classe *Material*, foi implementada uma primeira classe concreta que representa os materiais isotrópicos, a classe *IsotropcMaterial*.

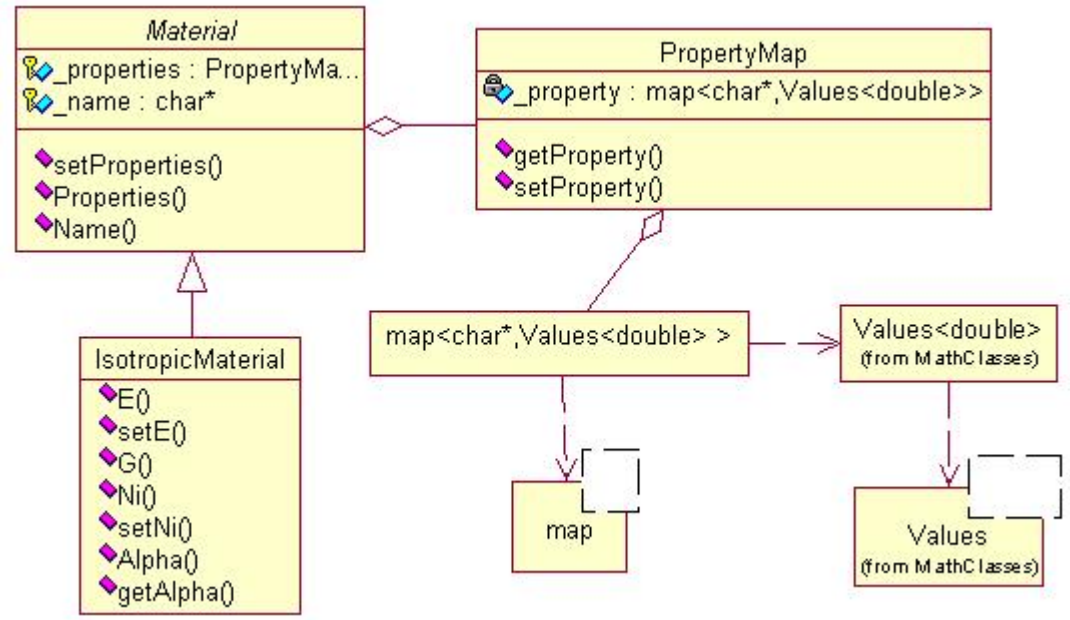

FIGURA 5.6 – Classes do componente Material Classes

5.1.2 O componente Kernel

Para melhor organizar as classes que formam o componente *Kernel*, este foi organizado em seis pacotes lógicos: *Domain Classes*, *Builder Classes*, *Factory Classes*, *DataBase Classes*, *Mesh Classes*, *DataManager Classes* e *Boundary Classes*. Cada pacote armazena uma ou mais classes de modelo ou de controle, organizadas segundo suas funções gerais dentro do *framework* (FIGURA 5.7).

Pacote *Domain Classes*

O pacote *Domain Classes* (FIGURA 5.8) contém a classe *Domain* responsável pela abstração dos dados ou valores específicos resultantes de uma formulação de análise, de um processo de dimensionamento ou de detalhamento utilizado para configurar um domínio ou necessários para a execução das rotinas de cálculo como: número de passos para processos iterativos, incrementos de carga, número de fatias de uma seção transversal, etc. São dados que devem ser informados pelo usuário de tais processos, ou sistemas, e que não fazem parte dos dados modelados pelo *framework,*

mas que não justificam a definição de novas classe de modelo na hierarquia de *Model.* Os dados são então armazenados e gerenciados por uma classe *Domain*, podendo ser representados facilmente por um ou mais pares: especificação x valor numérico.

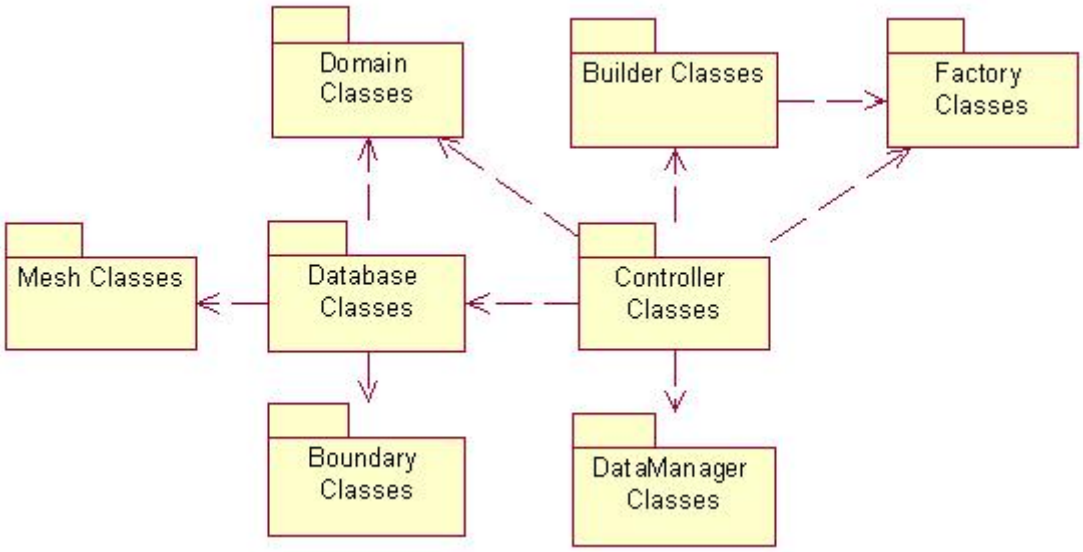

FIGURA 5.7 – Pacotes do componente *Kernel*

O preenchimento padrão destes dados pode ser realizado na criação de um objeto de domínio através do método *Initialize*. O método *Update* permite que os dados sejam alterados durante a utilização dos objetos, para usos específicos do sistema, ou a cada chamada do mesmo.

Pacote *Builder Classes*

No pacote *Builder Classes* é organizada a hierarquia da classe abstrata *Builder* (FIGURA 5.9), que implementa a construção de objetos complexos, segundo o padrão de mesmo nome, apresentado na seção 4.2.5. A classe *Builder* possui uma referência para um objeto do tipo *Filler* para possibilitar a construção dos objetos através da obtenção ou da leitura de dados de uma interface de arquivo ou de interfaces com usuários.

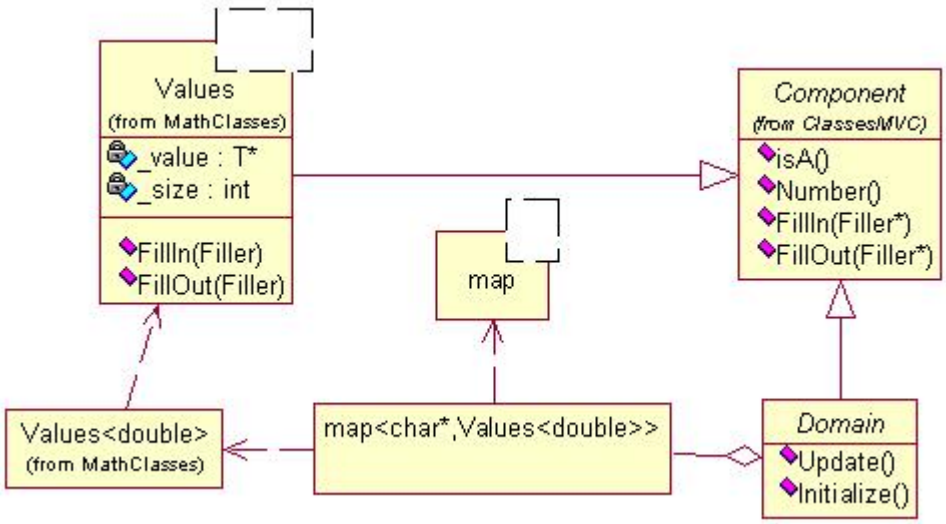

FIGURA 5.8 – Pacote *Domain Classes*

Na hierarquia da classe *Builder* foram instanciadas algumas classes concretas responsáveis por construir objetos complexos. Estes objetos podem ser elementos geométricos como *Sections*, *Shapes*, elementos unifilares criados a partir do refinamento de elementos de um modelo, objetos que encapsulam propriedades físicas do modelo (*Material*) ou elementos de condições de contorno (*Load*).

A classe *ShapeBuilder* é também abstrata e declara a interface para a geração de formas geométricas que, isoladamente ou em conjunto, formam uma seção transversal. Cada objeto do tipo *Element,* definido no pacote *Mesh Classes* (FIGURA 5.12), pode receber uma seção transversal, formando elementos do modelo geométrico sólido. Duas classes concretas foram implementadas para possibilitar a geração de seções geométricas genéricas; as classes *CircleBuilder* responsável por definir uma forma circular e a classe *PolygonBuilder* capaz de criar uma forma poligonal a partir da definição dos *n* vértices que definem seu perímetro.

A classe *Mesh Builder,* derivada da classe *Builder,* automatiza a geração de uma malha de elementos lineares obtidos a partir de entidades do modelo geométrico, inserindo-os em um novo modelo chamado de *REMModel* (modelo refinado). Para possibilitar novos refinamentos, ou seja novos conjuntos de elementos lineares obtidos

por divisões dos elementos lineares de um modelo, a classe *Mesh Builder* possui uma referência para a classe *Model* e uma referência para a classe fábrica *ViewFactory,* responsável por instanciar as representações gráficas. O modelo *REMModel* é utilizado para transferir dados geométricos, de materiais e de condições de contorno para os sistemas de análise e por isso é chamado também de modelo de análise ou *FEMModel,* em analogia aos sistemas estudados no capítulo 3.

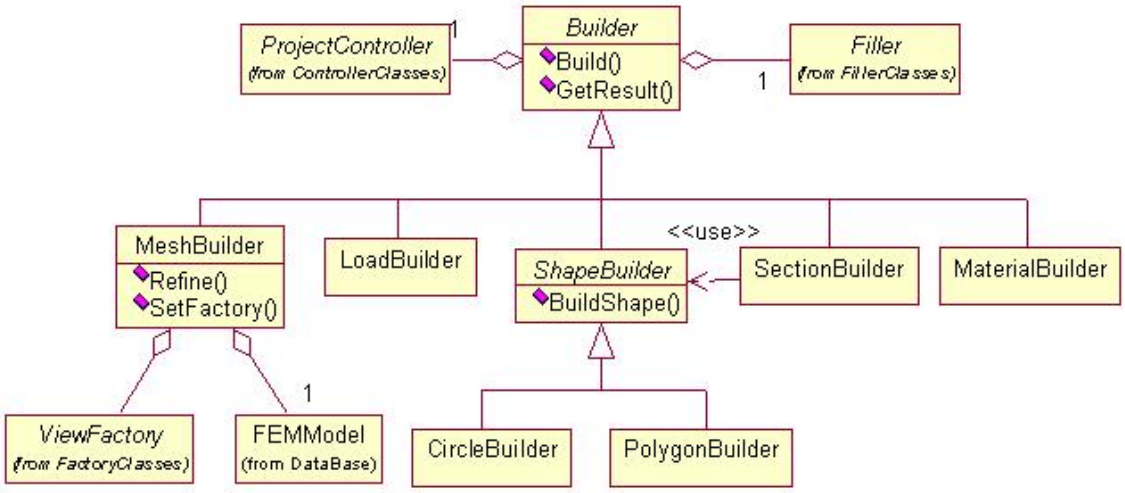

FIGURA 5.9 – O pacote *Builder Classes*

Pacote *Factory Classes*

O pacote *Factory Classes* apresenta classes que implementam o padrão *Abstract Factory* (item 4.2.5) para a criação de elementos relacionados. Neste pacote encontramos as classes *ViewFactory, BuilderFactory* e *AnalysisFactory* (FIGURA 5.10). Estas classes especificam uma interface de métodos fabrica (padrão *Factory Method*) os quais permitem a criação de objetos específicos correlacionados. Estes métodos permitem que o núcleo do *framework* faça chamadas aos objetos definidos por cada aplicação, sem que este conheça os objetos que são criados.

A classe *ViewFactory* especifica a interface de métodos para criação de objetos de visualização através de classes instanciadas no *hot spot* formado pela classe *View*. A

classe *AnalysisFactory* especifica métodos para a criação de objetos do tipo *DataManager* e *Domain,* utilizados para gerenciar e representar dados específicos de cada sistema de análise integrado através do *framework*. Os métodos da classe *BuilderFactory* retornam objetos do tipo *Builder* definidos e utilizados por cada aplicação para a criação dos objetos do tipo *Section, Shape, Material* e *Load* e ainda objetos do tipo *MeshBuilder* responsáveis por criar os elementos do modelo *REMModel*.

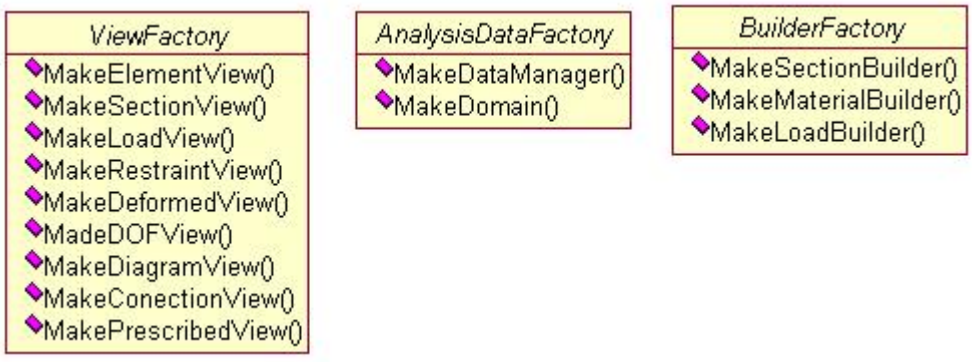

FIGURA 5.10 – Pacote *Factory Classes*

Classes fábrica concretas devem ser criadas nas aplicações desenvolvidas a partir do *framework*. Uma aplicação que implementa um sistema CAD para modelagem da estrutura precisa instanciar uma fábrica concreta, a qual é responsável por retornar os objetos que implementam a visualização dos dados do modelo tratado. Uma aplicação de integração que utiliza um ou mais sistemas externos é responsável por instanciar uma fábrica concreta do tipo *AnalysisDataFactory*. Da mesma forma uma aplicação que necessite definir o modo de criação de objetos do tipo *Material*, *Section* ou *Load* deve instanciar uma fábrica concreta do tipo *BuilderFactory*.

Pacote *DataManager Classes*

A classe *DataManager* define uma interface para métodos de persistência de dados dos elementos do modelo. Estes métodos permitem obter, organizar e, se necessário, processar os dados tanto para carregar elementos do modelo a partir de sistemas de persistência quanto gravá-los nestes sistemas. Esta classe e a classe *Filler*,

definem dois *hot spots* que implementam a interface de arquivos. Os objetos de modelo são ordenados pelos métodos definidos em sua interface para gerar uma formatação adequada à transferência de dados entre o *framework* e os sistemas externos integrados por este.

A classe *DataManager* possui uma referência para um objeto da classe *Project*, através da qual obtém acesso aos objetos de modelo, sem violar o encapsulamento de dados. A classe *Project* é uma classe que armazena, direta ou indiretamente, todos os objetos que representam uma estrutura e disponibiliza uma interface de métodos de acesso a estes elementos. Através dela e dos métodos de acesso implementados em cada classe de modelo, a classe *DataManager* é capaz de obter e manipular os dados necessários para a transferência de dados entre sistemas. Esta classe possui ainda duas referências para objetos do tipo *Filler*: um destinado à manipulação de documentos para leitura de dados e outro para a manipulação de documentos de saída de dados. Desta forma, para um mesmo ambiente, ou domínio de transferência de dados, é possível tratar de forma independente e diferenciada dois tipos de formatação e dois sistemas de persistência: um para leitura e outro para gravação. Os relacionamentos da classe *DataManager* são mostrados na FIGURA 5.11.

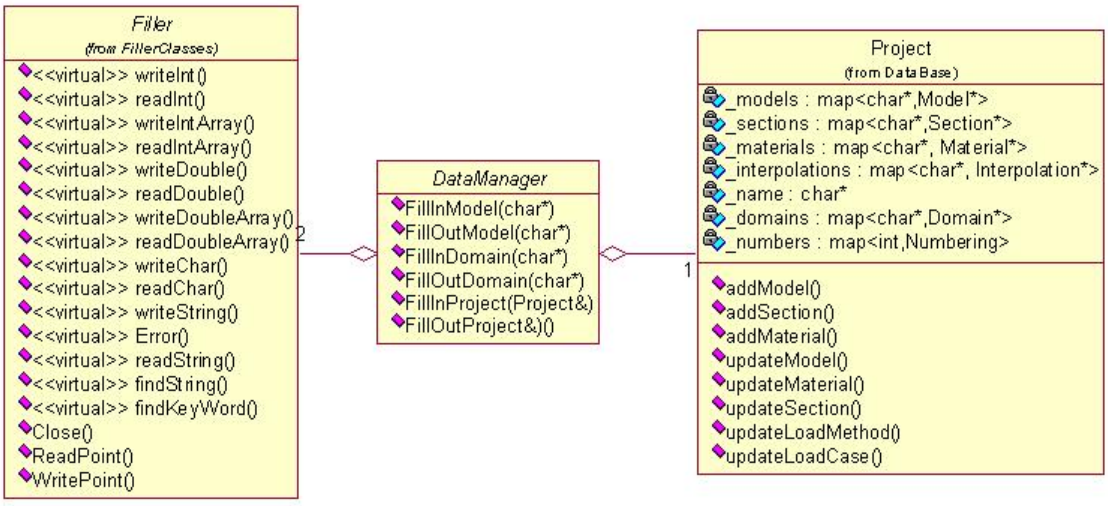

FIGURA 5.11 – Pacote *DataManager*

Pacote *Mesh Classes*

Outro pacote do componente *Kernel* é o pacote *Mesh Classes*. Este pacote organiza as classes de elementos de modelo, que fazem a abstração dos elementos unifilares de uma estrutura reticulada discretizada. Neste pacote estão as classes *Element*, *Node*, *Bar*, *ConectionElement*, *ConectionRule*. Estas classes são utilizadas para gerar o modelo geométrico, onde os elementos estruturais são definidos, e os modelos refinados para a análise, podendo comportar ainda outros modelos para representar a estrutura em outras etapas do processo de projeto. Por definição de projeto, o modelo geométrico é aquele de mantém os dados obtidos dos demais modelos, adotados como solução para a estrutura. Assim, o sistema de integração pode recuperar dados de um ou mais modelos e, após sua verificação e validação, os dados adotados como solução para a estrutura são retornados, direta ou indiretamente, para o modelo geométrico. Por exemplo: para a análise, um elemento do modelo geométrico pode ser discretizado em *n* elementos no modelo refinado. Após a etapa de análise, executada pelos sistemas externos, os resultados da análise são recuperados pelo *framework*. Devido à relação entre o elemento geométrico e o conjunto de elementos discretizados, obtidos a partir deste, é possível selecionar o esforço máximo encontrado dentro do conjunto, retornando valor e posição deste esforço para o elemento geométrico original. Este valor pode então, ser formatado e repassado, na etapa de dimensionamento, para os sistemas específicos de dimensionamento integrados no processo.

A classe abstrata *Element* fornece a interface para todos os objetos a serem inseridos em um modelo da estrutura reticulada. Desta classe abstrata foram derivadas três classes de elementos: *Node*, *Bar* e *ConectionElement*. A classe *Element* e seus relacionamentos são apresentados na FIGURA 5.12. A conectividade entre diferentes elementos do modelo (nós, barras, elementos de conexão) é garantida através de um atributo em cada elemento que armazena os identificadores de todos os demais elementos a ele relacionados.

A classe *Bar* faz a abstração dos elementos lineares do modelo. Através dela são representados os elementos estruturais lineares como pilares, vigas e elementos de treliça, etc., assim como os elementos dos modelos refinados. A classe *Node* representa

os pontos inicial e final de cada elemento do tipo *Bar,* além dos pontos de interseção entre estes elementos. A classe *ConectionElement* faz a abstração de elementos utilizados para modelar uma ligação entre os elementos do modelo estrutural. Uma ligação deve estar sempre associada a elementos do tipo *Node*. O tratamento geométrico dos objetos do tipo *Bar* que chegam ao objeto *Node*, assim como as regras de relacionamento entre os demais elementos que formam a ligação, são definidas por um objeto de uma classe derivada da classe *ConectionRules*. Esta classe é uma classe abstrata que define uma interface para os métodos de relacionamento ou de associação entre os elementos de ligação e os objetos do tipo *Bar*. Como as regras de associação podem variar conforme o tipo de ligação e o tipo de elementos conectados por ela, para permitir a variação destas regras de ligação, foi adotado o padrão *Strategy* (item 4.2.5).

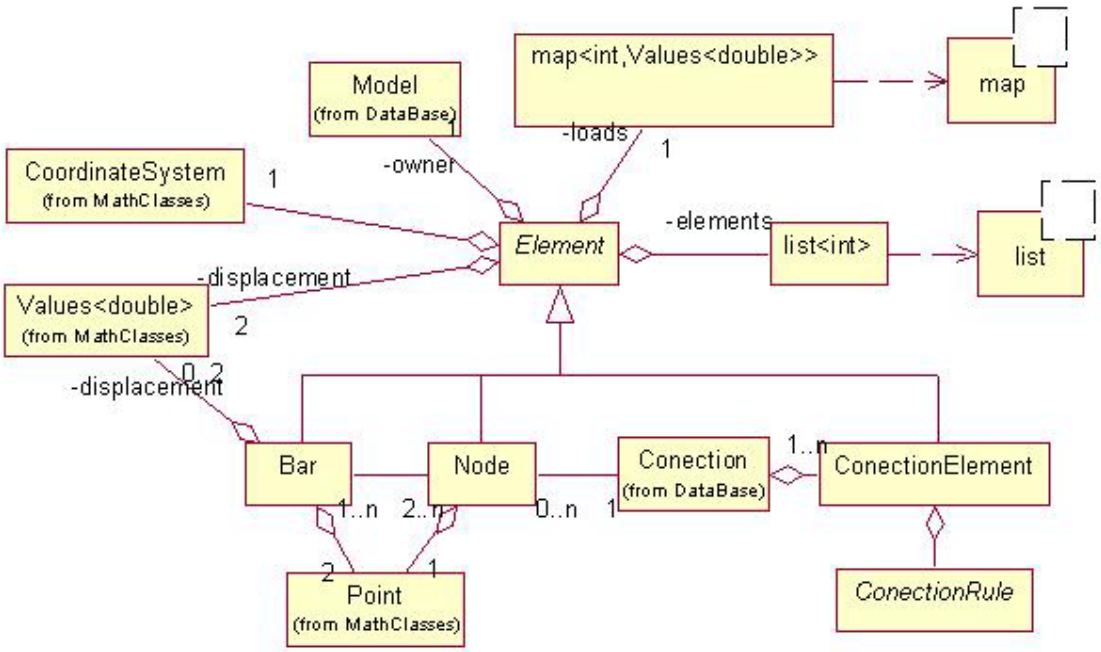

FIGURA 5.12 – Pacote *Mesh Classes* e relacionamentos da classe *Element*

O padrão *Composit* (GAMMA et al, 2000) foi inicialmente escolhido (LEITE e FRANCO, 2004) para representar este tipo de relacionamento onde um elemento está relacionado a vários outros objetos do mesmo tipo. Porém no desenvolvimento do sistema a utilização das listas de identificadores juntamente com classes contêineres se mostrou mais adequada, facilitando o processo de atualização de dados no modelo.

Pelas características do domínio tratado, um mesmo elemento pode ser referenciado por vários outros elementos, havendo referências cruzadas. Observou-se que armazenar, em cada objeto, referências diretas aos elementos relacionados a ele, seria dispendioso tanto em termos de memória quanto em termos de tempo de processamento para buscas. Assim, a estruturação dos elementos utilizando referências por meio de chaves de identificação, mantendo todos os objetos de um modelo armazenados em classes que denominamos de classes de banco de dados, permitiu simplificar a estrutura dos elementos e os processo de busca e de alterações. Este tipo de relacionamento por identificadores e não por ponteiros ou referências foi baseado na filosofia apresentada para o ambiente OOPAR sintetizada pelo item 3.9.2 da presente tese. Uma barra, então, não armazena diretamente suas coordenadas inicial e final, nem ponteiros ou referências para seus nós de extremidade. Uma barra armazena os identificadores de seus nós, com os quais obtém, da classe contêiner à qual pertence, seus nós de extremidade, solicitando dos mesmos sua posição sempre que necessário.

Pacote *Boundary Conditions*

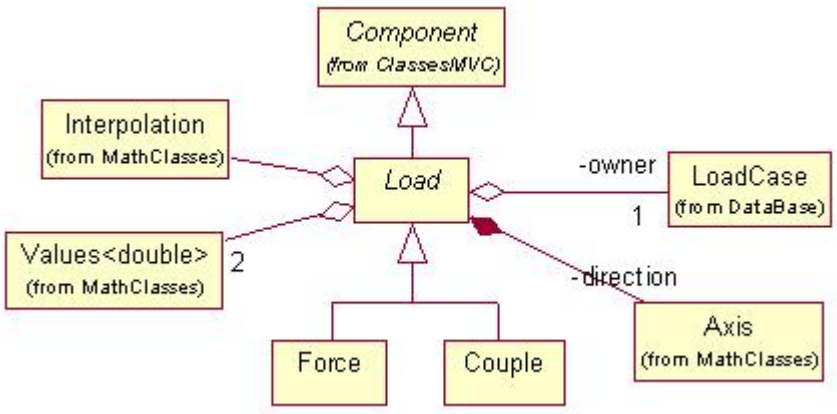

FIGURA 5.13 – Pacote *Boundary Conditions*

O pacote *Boundary Conditions* (FIGURA 5.13) reúne as classes que fazem a abstração das condições de contorno aplicadas ao modelo refinado, como as classes de carregamento: *Load*, *Force* e *Couple*. As demais condições de contorno como:

restrições nodais, deslocamentos prescritos, variações de temperatura e deformações iniciais, são abstraídas por atributos armazenados nas classes de banco de dados que representam o modelo refinado da estrutura reticulada (*REMModel*).

As cargas podem variar em valor (inicial e final), aplicação concentrada ou distribuída ou em direção de aplicação (eixos globais da estrutura). Para definir o eixo de aplicação a classe *Load* possui um atributo do tipo *Axis*. Os pontos de aplicação e os valores inicial e final da carga são definidos por listas de valores numéricos, objetos *Values*. Assim, os pontos de aplicação não são armazenados nos objetos *Load*, mas são obtidos a partir das coordenadas dos elementos sobre os quais a carga está aplicada e estão associados às distâncias de aplicação armazenadas nos objetos *Values*. Assim, cargas distribuídas podem ser aplicadas em todo o comprimento de objetos do tipo *Bar*, ou em apenas parte destes objetos. Uma carga distribuída é associada a um objeto de interpolação polinomial (classe *Interpolation* do componente *Math Classes*) permitindo uma variação da carga ao longo do trecho de aplicação. Os carregamentos pontuais podem ser aplicados sobre objetos *Node* e *Bar*.

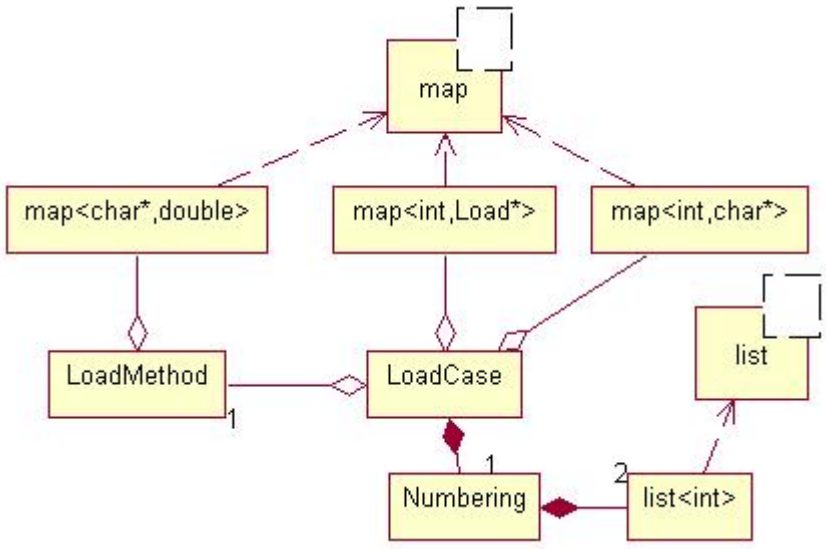

FIGURA 5.14 – Classe *LoadCase* e *LoadMethod* do pacote *DataBase Classes*

Vários carregamentos podem ser agrupados sob um caso de carregamento abstraído por um objeto da classe *LoadCase,* semelhante ao proposto pelos sistemas

Modelador 3D e derivados deste, e pelo projeto INSANE. Os coeficientes aplicados sobre cada carregamento podem variar conforme o tipo de carga: permanente, variável, carga de vento, etc. As cargas podem então, ser classificadas quanto ao tipo de carregamento. Os coeficientes aplicados sobre cada tipo de carregamento é definido por um objeto da classe *LoadMethod,* semelhante ao proposto pelo projeto INSANE através da classe *LoadCombination*. Estas duas classes, *LoadCase* e *LoadMethod*, são classificadas como classes de banco de dados, possuindo dicionários que armazenam cargas, classificações de cargas e coeficientes de majoração de cargas (FIGURA 5.14).

Pacote *DataBase Classes*

O pacote *DataBase Classes* (FIGURA 5.15) reúne as classes que organizam os dados do modelo, armazenando os objetos de dados em classes contêineres. São utilizados como classes contêineres: listas encadeadas ou dicionários. As classes de banco de dados armazenam e gerenciam informações, provendo métodos de busca e de acesso aos objetos gerenciados. Os dicionários são formados por pares de chaves lógicas e ponteiros ou referências para classes de dados. O tipo de chave de acesso varia conforme o tipo de elemento armazenado, podendo ser o nome dado ao elemento para seções, materiais e modelos, ou o identificador numérico do objeto para nós, barras, cargas e casos de carregamento. A chave de acesso é responsável por identificar de maneira única um objeto dentro de uma classe de banco de dados. Através desta identificação é possível verificar a existência de um dado, recuperar informações dos elementos armazenados, inserir novos elementos, excluir elementos ou alterar elementos já inseridos no modelo, realizar verificações de consistência e validações.

Por especificação de desenho, um projeto foi definido como um conjunto de modelos que representam uma estrutura reticulada em diferentes contextos. São considerados modelos: a malha que representa a geometria unifilar de uma estrutura, os elementos refinados de um sistema, os elementos de detalhamento de uma estrutura, etc., além das demais informações necessárias para caracterizar cada modelo. Dados que podem ser utilizados por mais de um modelo são armazenados no projeto, enquanto

cada modelo armazena as chaves de acesso a estes dados, de maneira semelhante à implementação de listas de identificadores na classe *Element*.

A classe *Project (*FIGURA 5.16) faz a abstração de um projeto. Esta classe possui dicionários de modelos, dicionários de materiais, dicionários de seções, de domínio e de numerações. Todas as classes armazenadas pelas classes de banco de dados, e estas últimas, são derivadas da classe *Component* do componente *Classes MVC*. Cada uma destas classes implementa seu método de preenchimento (*FillIn*) e seu método de gravação (*FillOut*). Estes métodos são responsáveis por permitir a persistência dos objetos complexos gerenciados pelo *framework* e por recompor estes objetos a partir dos sistemas de persistência. São considerados sistemas de persistencia os documentos ou conjunto de documentos (como tabelas de um banco de dados) capazes de armazenar os dados dos objetos de um modelo após a execução de uma aplicação. Durante a execução de uma aplicação os objetos de uma estrutura são carregados pelas classes do pacote *DataBase*, e manuseados exclusivamente através destas classes, para garantir a consistência dos dados ao longo de todo o processo.

Além da classe *Project*, foram definidas as seguintes classes banco de dados: *Model* e *LoadCase* e *LoadMethod*. A classe *Model* (FIGURA 5.16) armazena os elementos da malha (*Node, Bar, ConectionElement*) e as condições de contorno aplicadas à mesma, carregamentos, restrições nodais, deslocamentos prescritos, etc. Os carregamentos são armazenados na classe *Model* através da classe *LoadCase*. Um modelo pode conter mais de um caso de carregamento armazenando ponteiros para a classe *LoadCase* em um dicionário. Um objeto do tipo *LoadCase* armazena por sua vez uma referência para o modelo onde foi inserido. As demais condições de contorno são armazenadas em um modelo através de dicionários. Estes dicionários armazenam uma estrutura que representa a condição de contorno, geralmente um objeto da classe *Values*, vinculado a uma chave de acesso, que é a identificação do elemento ao qual a condição de contorno está associada. Assim uma restrição ou uma liberação não é inserida diretamente no elemento, mas é inserida em uma estrutura independente, pertencente à classe *Model*. Esta solução visa proporcionar uma busca mais rápida por condições de contorno, que geralmente são aplicadas a um número restrito de elementos se comparado com o total de elementos presentes em um modelo.

A classe *LoadCase* possui dois dicionários principais: um para armazenar as cargas aplicadas ao modelo e outro para descrever o tipo de carga aplicada. A especificação do tipo de carga é necessária para a correta criação do objeto de carregamento quando um projeto é recuperado através dos dados de persistência. Os demais elementos do modelo não necessitam desta especificação porque cada tipo de elemento (nó, barra, ligação, seção, etc.) é representado por um objeto de visualização específico. Porém, em um objeto *LoadCase* são armazenados todos os tipos de cargas, sendo necessário identificar cada tipo armazenado para a correta recuperação de dados persistentes e a correta vinculação com objetos de visualização, que variam conforme o tipo de carga armazenada (*Force* ou *Couple*).

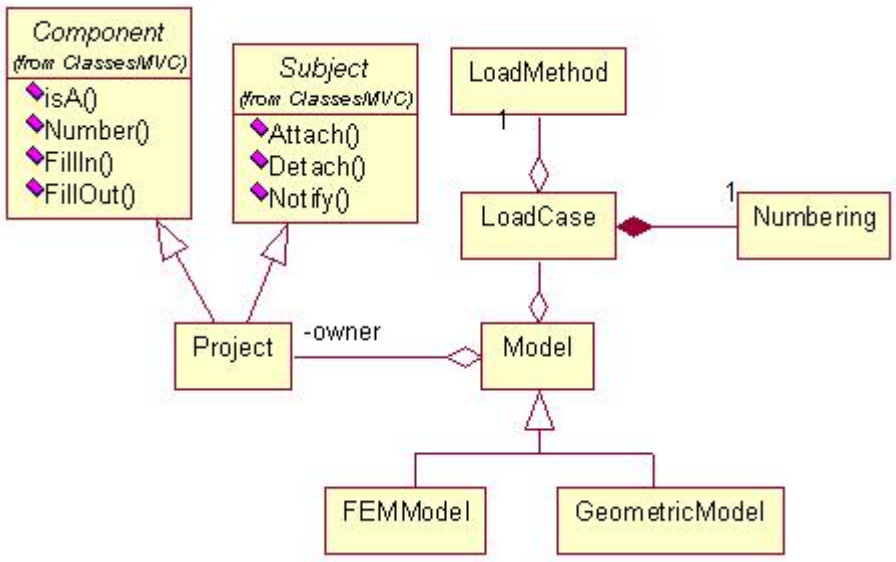

FIGURA 5.15 – Classes do pacote *DataBase Classes*

Também no pacote *DataBase Classes* foram inseridas as classes *Numbering* e *LoadMethod.* A classe *Numbering* é responsável por gerenciar um conjunto de números utilizados para identificar de forma ordenada um conjunto de elementos. Objetos da classe *Numbering* são utilizados para gerenciar a numeração de nós, de barras, de ligações, a de carregamentos e de casos de carregamentos. Os objetos de numeração são armazenados em dicionários, dentro da classe de banco de dados que os utiliza

(FIGURA 5.16). A classe *LoadMethod* armazena os coeficientes aplicados sobre cada tipo de carga definida no projeto por meio de um dicionário, que relaciona a identificação do tipo de carga a um coeficiente numérico.

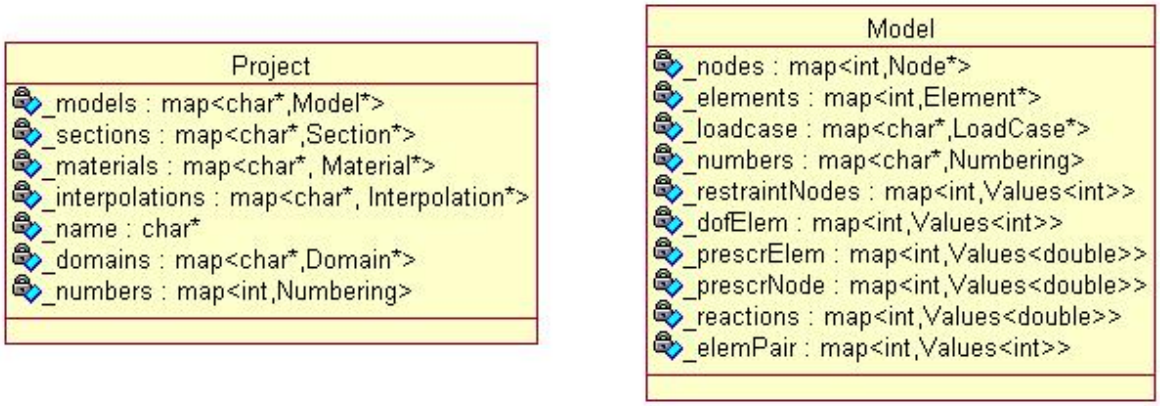

FIGURA 5.16 – Dicionários das classes *Model* e *Project*

Pacote Controller Classes

O pacote *Controller Classes* é formado pela classe *ProjectController* (FIGURA 5.17), responsável por gerenciar a criação ou edição de objetos do modelo, pela associação entre estes objetos e suas respectivas visualizações, e pela coordenação dos fluxos de execução do aplicativo. Esta classe transmite à classe *Project* as solicitações do usuário, além de instanciar os objetos responsáveis por formatar os dados do modelo de acordo com as exigências dos sistemas a serem integrados pela aplicação. A classe *ProjectController* possui referências para objetos das classes abstratas que formam os *hot spots* do *framework*. Através destas referências a classe *ProjectController* é capaz controlar o fluxo de informações entre a aplicação e as classes do *framework*, sem conhecer detalhes dos objetos instanciados. As classes concretas definidas na aplicação são instanciadas a partir dos *hot spots* do *framework* para interface de arquivos e interface de usuários (*DataManager* e *Filler*), representação gráfica (*View*), construção de elementos (*ViewFactory*, *BuiderFactory*, *AnalysisDataFactory* e *Builder*) e configurações do domínio (*Domain*).

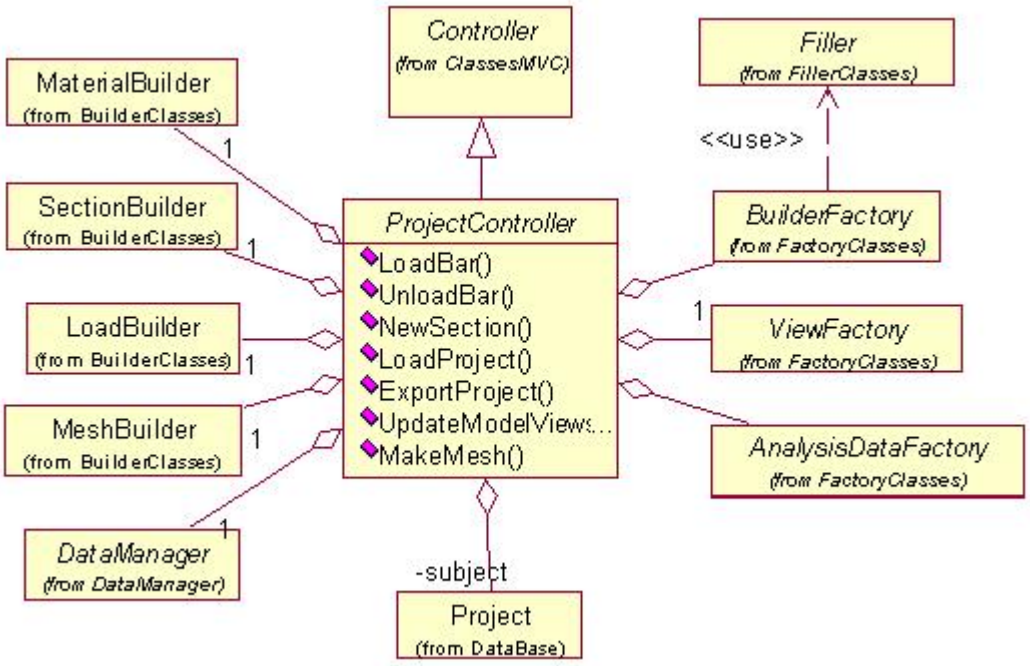

FIGURA 5.17 – Pacote *Controller Classes*

5.2 *Hot spots* do *framework* REMFrame

Através da análise de domínio foi possível identificar um conjunto de dados gerais relacionados a projetos de estruturas reticuladas, comuns a diversos tipos de estruturas, tais como: os dados que representam uma estrutura unifilar (barras e nós), a necessidade de representar condições de contorno, dados para armazenar propriedades físicas dos materiais, dados sobre as seções transversais, etc. O conjunto completo de dados definidos para o domínio permitiu definir abstrações capazes de representar todo um conjunto de tipos de estruturas e por isso presentes nos *frozen spots* do *framework*. Entre os *frozen spot* podemos citar as classes *Bar, Node, Isometric Material, Section, Force, Couple, GeometricModel, FEMModel, Project, ProjectController* e todas as classes matemáticas. Estas classes foram utilizadas sem alterações para representar aquelas estruturas com as quais o grupo CADTEC trabalhou anteriormente. Da mesma

forma foi possível representar outras estruturas processadas pelos diferentes sistemas de análise e de dimensionamento integrados pelas aplicações desenvolvidas com o *framework*. Estas estruturas foram incluídas como estudos de caso na utilização dos sistemas de integração apresentados no Capítulo 6.

Outros tipos de dados, ou de representações, são específicos das aplicações. Para estes dados foram adotados alguns mecanismos de flexibilização através da aplicação de padrões para a estruturação de classes e definição de relacionamentos. Estes dados são representados, então, por classes que devem ser definidas em cada aplicação, através dos *hot spots* definidos pelo *framework*. Constituem *hot* spots do *framework* REMFrame as seguintes classes: *ViewFactory, BuilderFactory, AnalysisDataFactory, View*, *Filler*, *DataManager*, *Domain* e *Builder.*

As características específicas dos sistemas externos integrados no processo através do *framework* são representadas por uma classe de configuração de domínio de aplicação, a classe *Domain*. Esta classe faz a abstração dos dados necessários para o processamento destes sistemas, sem alterar a arquitetura proposta para o núcleo do *framework*. Assim, a classe abstrata *Domain* permite que os dados requeridos por cada sistema externo sejam modelados por classes concretas derivadas desta e definidas nas aplicações. O núcleo do *framework* pode, então, manipular estes dados e informá-los aos sistemas externos sem conhecer os mesmos.

A interface da aplicação com os sistemas externos varia, necessariamente, com a formatação dos dados requeridos por cada sistema utilizado no processo. Assim o desenvolvimento de aplicações deve definir tanto a formatação dos dados para a transferência de informações, quanto os mecanismos utilizados para a leitura e gravação de dados. Para permitir a flexibilização do processo de transferências de dados, ou seja, a alimentação e/ou fluxo da dados entre aplicações, sistemas externos e usuários, foram definidos dois *hot spots*: classes *Filler* e *DataManager*. Estes dois *hot spots* permitem que a importação e a exportação de dados sejam definidas de forma independente em cada aplicação desenvolvida. Para a flexibilização da representação gráfica dos dados do modelo estrutural, foi definida a classe abstrata *View.*

Além de classes concretas, foram implementadas classes abstratas que disponibilizam interfaces para a criação de objetos. Estas classes constituem importantes *hot spots* por permitir que os objetos especializados sejam definidos pelas aplicações, mas criados e manipulados em rotinas definidas no núcleo do *framework*. São *hot spots* de criação de objetos, ou de uma família de objetos, as classes *ViewFactory, BuilderFactory, AnalysisDataFactory e Builder.*

De maneira geral, a estrutura de classes implementada busca oferecer uma funcionalidade básica, capaz de ser aproveitada sem alterações pelas aplicações geradas a partir do *framework* e ainda permitir a especialização destes sistemas através da derivação de novas classes concretas utilizando-se para isso os *hot spots* pré-definidos. Desta forma o *framework*, em seu estágio atual de desenvolvimento, apresenta características de *grey box* com utilização de alguns elementos através das funcionalidades já implementadas pelo *framework* e de outros por derivação de novas classes concretas a partir das classes abstratas existentes nos *hot spots*.

5.3 Casos de uso

A funcionalidade de um sistema pode ser definida pelo conjunto de resultados que o mesmo é capaz de gerar para seus usuários ou atores. Cada resultado mensurável obtido através de um fluxo de ações desempenhadas pelo sistema é definido como um caso de uso do mesmo (FURLAN, 1998 e QUATRANI,1998).

O *framework* REMFrame tem implementadas as seguintes funcionalidades gerais: definição do modelo geométrico, inserção de propriedades físicas e geométricas nos elementos do modelo, definição de condições de contorno, geração do modelo discretizado, transferência de dados para sistemas externos, importação e representação de resultados. Estas funcionalidades gerais foram implementadas pelos seguintes casos de uso:

Insert Bar: Inserção de barras no modelo;

Remove Bar: Supressão de barras do modelo;

Make Mesh: Construção do modelo discretizado;

New Section: Definição de uma seção transversal;

New Material: Definição de um material;

Apply Section: Aplicação de uma seção transversal a elementos do modelo;

Apply Material: Aplicação de um material a elementos do modelo;

Add Load: Definição de um carregamento vinculado a elementos do modelo;

Add Restraints: Definição e aplicação de restrições nodais;

Load Project: Leitura de dados de um projeto, a partir do sistema de persistência;

Save Project: Gravação de dados de um projeto em um sistema de persistência.

Export Data: Formatação e exportação de dados e acionamento de sistemas externos;

Import Data: Importação de dados de sistemas externos para modelagem automática da estrutura;

Draw Model: Apresentação da visualização gráfica de um modelo do projeto;

Read Deformation: Leitura dos deslocamentos resultantes da análise e representação da deformada da estrutura;

Read Results: Leitura dos esforços resultantes da análise e representação dos gráficos correspondentes.

A seguir passamos a uma descrição sintetizada do fluxo principal de cada um dos casos de uso definidos. Os diagramas de seqüência apresentados mostram fluxos simplificados, onde apenas as ações principais foram apresentadas, visando identificar principalmente a transferência de controle de fluxo entre o *frozen* spot e os *hot spots*.

5.3.1 Casos de Uso *Insert Bar* e *Remove Bar*

Os casos de uso *Insert Bar* e *Remove Bar* são os principais fluxos, que permitem a definição dos elementos lineares do modelo geométrico e dos demais modelos. Estes elementos podem ser definidos pelo usuário de forma interativa.

O lançamento da estrutura unifilar é feito através do caso de uso *Insert Bar* (FIGURA 5.18). A partir de um objeto geométrico do tipo *Line* a classe de controle define e insere no modelo dois objetos do tipo *Node* nas coordenadas dos pontos inicial e final, e um objeto do tipo *Bar* tendo como nós de extremidade os objetos *Node* criados.

Durante este processo de inserção de um elemento linear no modelo estrutural, são realizadas todas as verificações de consistência do modelo. Entre as verificações realizadas pelo controlador estão: a verificação da existência prévia de nós ou barras nas posições definidas para a criação de um novo elemento e o cálculo de interseções entre elementos lineares. Para realizar estas verificações um objeto controlador utiliza os métodos definidos pelas classes matemáticas *Line* e *Matrix*. A classe *Line* define os métodos *Intersection* e *Overload,* que por sua vez utilizam o método *SolveLinearSystemLU* definido na classe *Matrix*. As interseções entre objetos do tipo *Bar* podem ser verificada ou não. Caso sejam verificadas elas podem ser permitidas ou não. Estas configurações de permissão ou restrição são definidas pelo usuário em tempo de execução para o projeto, podendo ser alteradas ao longo do lançamento da estrutura. Caso as interseções entre objetos do tipo *Bar* sejam verificadas e permitidas, um objeto do tipo *Node* é criado com as coordenadas do ponto de interseção retornado pelo método *Intersection*. O identificador deste objeto é então adicionado em ambas as barras como um nó intermediário, sem dividir as barras originais. Se a interseção é permitida e não verificada, a barra é criada normalmente, mas não é criado um nó no ponto de interseção entre os elementos do tipo *Bar*. Caso a interseção não seja permitida, ou caso haja uma sobreposição entre elementos, o objeto *Bar* não é criado. Caso um objeto do tipo *Bar* seja criado com extremidade coincidente com a

extremidade de outro objeto *Bar*, o objeto *Node* existente passa a fazer a ligação entre as duas barras.

Ao serem inseridas barras no modelo estrutural, seu nó inicial e nó final são criados e numerados automaticamente através do uso de objetos do tipo *Numbering* que realizam o controle da numeração dos diversos tipos de objetos inseridos em um modelo. Para a ordenação dos nós de uma barra são utilizados operadores definidos na classe *Point*.

Ao ser inserido um objeto do tipo *Bar*, o objeto fábrica da classe de controle é acionado para retornar um objeto de visualização para o elemento linear. Este objeto fábrica deve ser definido pela aplicação, através do *hot spot ViewFactory*. O objeto de visualização criado pelo método fábrica também é definido pela aplicação, através do *hot spot View*, ficando a seu encargo a implementação dos métodos *Draw* e *Erased*. Quando cada elemento da malha é criado, um observador derivado da classe *View* é inscrito em sua lista de observadores, dando início ao mecanismo de propagação de mudanças.

Quando um objeto *Element* é inserido, atualizado ou retirado do modelo, seu método *Notify* é acionado, desencadeando o processo de atualização das suas visualizações. O método *Notify* faz a chamada ao método *Update* do objeto observador. Este método por sua vez aciona os métodos *Draw* ou *Erased* do objeto de visualização. Através deste processo o *frozen spot* controla o fluxo geral do caso de uso deixando para as aplicações a criação dos objetos de visualização e a definição de sua representação gráfica.

O fluxo deste caso de uso mostra bem como a inversão de controle foi obtida no REMFrame. O desenho da representação gráfica não é acionado pelo usuário, mas pelas classes do *framework*. O núcleo do *framework* não tem conhecimento de como esta representação é realizada. As classes do modelo apenas acionam seus observadores e disponibilizam seus dados para estes. (FIGURA 5.18)

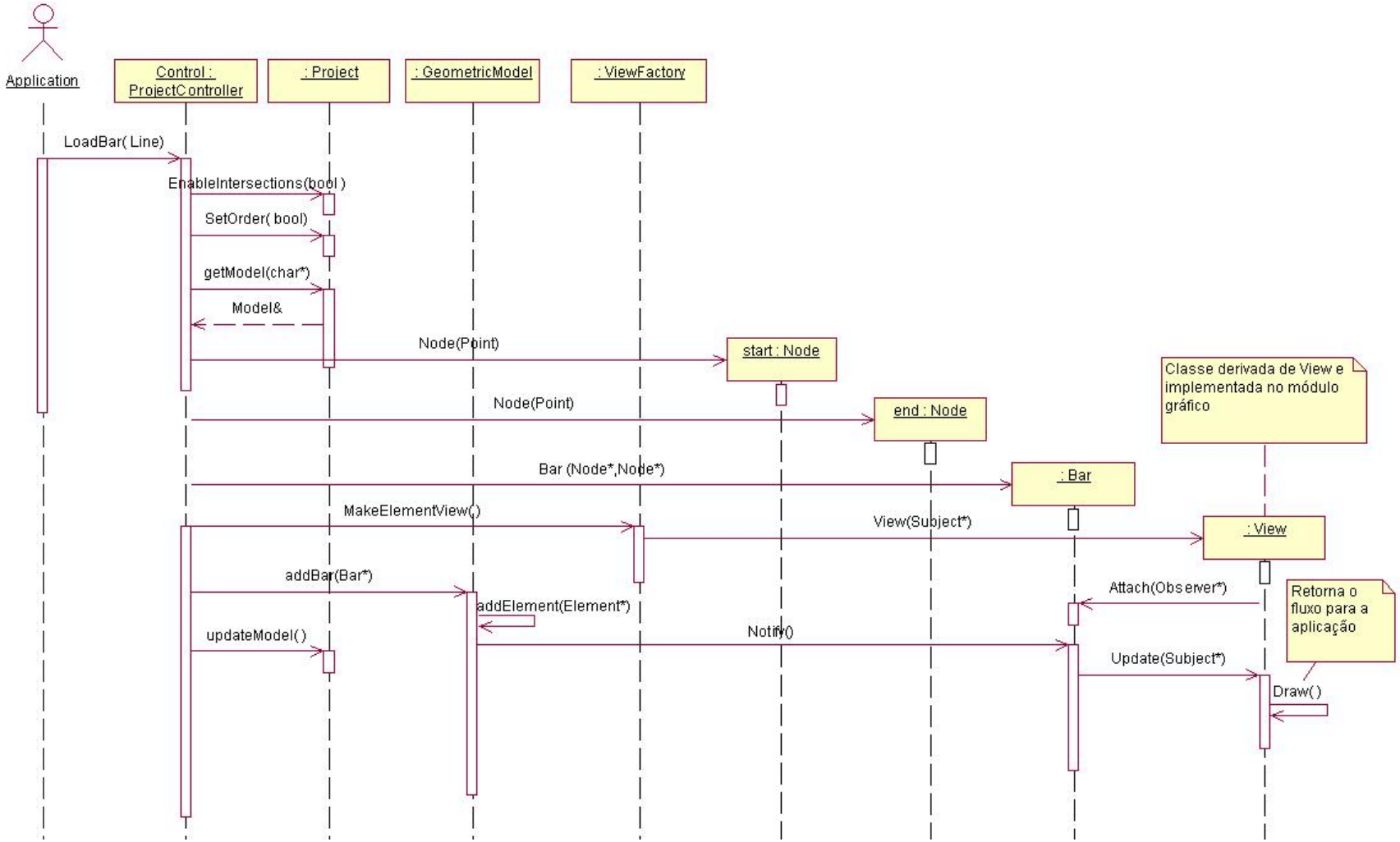

FIGURA 5.18 - Diagrama de seqüência: Caso de uso *Insert Bar*

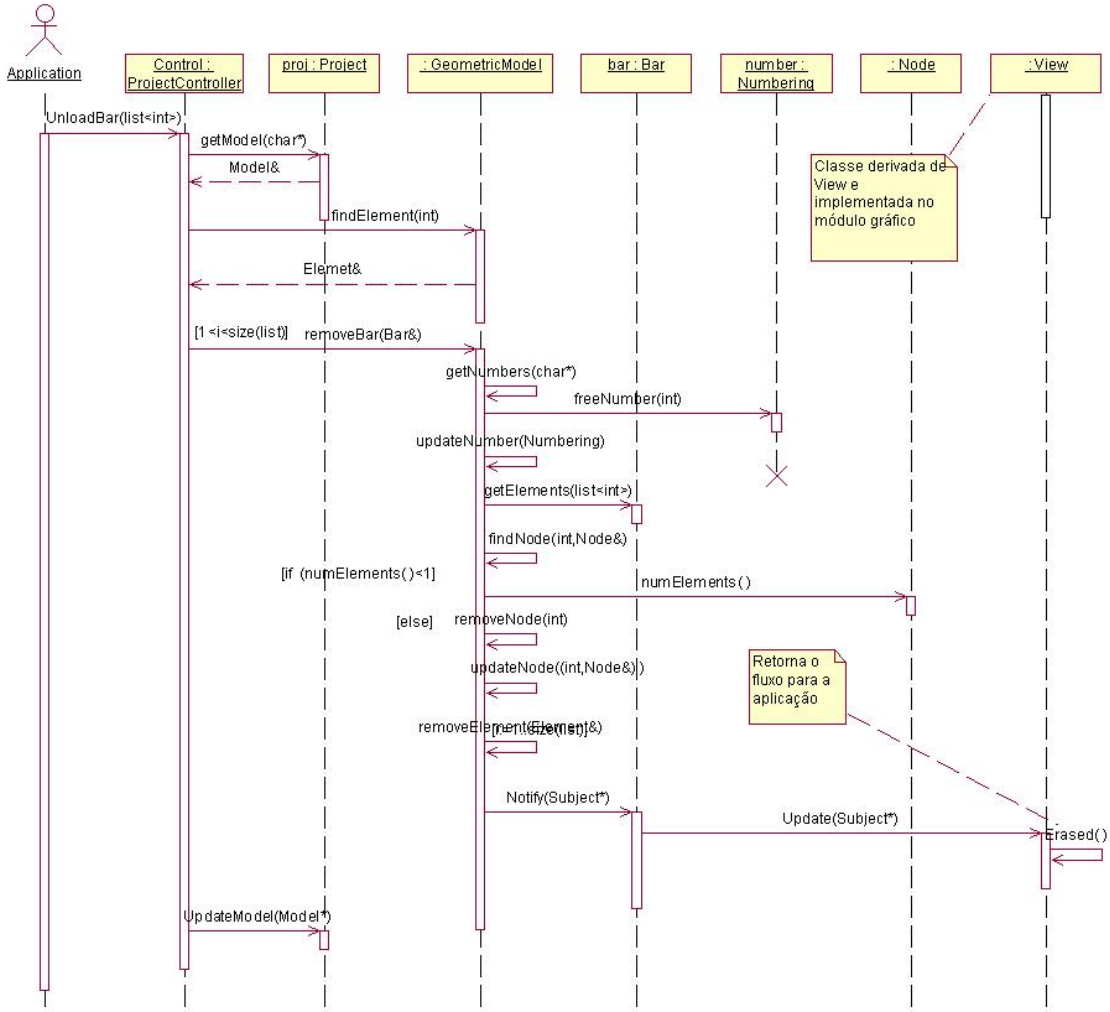

FIGURA 5.19 – Diagrama de seqüência: caso de uso *Remove Bar*

Para o caso de uso *Remove Bar* (FIGURA 5.19), ao ser retirado um elemento do modelo, sua numeração é recuperada pela classe *Numbering* possibilitando sua reutilização na inserção de novos elementos. Todos os nós vinculados ao elemento retirado são atualizados para excluir a vinculação com o mesmo. Caso o único elemento vinculado a um nó seja o elemento excluído, o nó é retirado do modelo. Após as verificações de consistência, o objeto é retirado do modelo e sua visualização é removida da representação gráfica através do método *Erased,* implementado na classe de visualização definida pela aplicação.

Algumas decisões de projeto foram tomadas, visando simplificar o controle de alterações no modelo e, a edição de um elemento inserido em um modelo sofre algumas restrições. É permitida a edição de propriedades físicas e geométricas que não geram alterações nos demais elementos do modelo como: definição de material e de seção transversal, porém, a alteração direta da posição espacial de um elemento, não é permitida. Para que uma barra seja alterada ela é inicialmente retirada do modelo e transformada em um elemento geométrico, um objeto do tipo *Line*. As alterações geométricas são, então, efetuadas sobre este objeto. Depois de alterada, a linha é transformada em um objeto do tipo *Bar* sobre o qual são efetuadas as validações necessárias, assim como as verificações de consistência. Assim a edição de elementos é realizada através dos casos de uso *Remove Bar* e *Insert Bar* utilizados em conjunto.

5.3.2 Casos de uso *Make Mesh*

Para representar o modelo estrutural podem ser utilizados um modelo geométrico e um ou mais modelos discretos. Para permitir uma correlação entre os elementos destes últimos com os elementos do modelo geométrico, tornou-se necessário implementar uma rotina que gera uma malha de elementos lineares sobre os elementos do modelo geométrico, permitindo a divisão de cada elemento do modelo geométrico em *n* elementos no modelo para a análise. A relação entre o elemento original e cada elemento gerado é armazenada no modelo discreto. Esta relação permite que os dados resultantes da análise possam ser organizados e trabalhados em grupos, obtendo-se valores de referência para o elemento geométrico, que deu origem a um conjunto de elementos discretos.

No caso de uso *Make Mesh* (FIGURA 5.20) a classe de controle recebe a solicitação de refinamento de um modelo discreto, identificado através de sua chave de armazenamento na classe de banco de dados. Os demais parâmetros repassados definem os elementos do modelo que devem ser refinados e o número divisões a ser aplicado sobre os elementos originais. O material e a seção transversal do elemento original são aplicados automaticamente aos elementos discretos criados. A classe *MeshBuilder* ao receber o objeto fábrica *ViewFactory* é capaz de vincular a cada elemento gerado

através do método *MakeMesh*, a visualização definida através do *hot spot* de visualização. A vinculação entre o elemento linear e sua visualização é realizada de maneira semelhante ao apresentado no caso de uso *Insert Bar*.

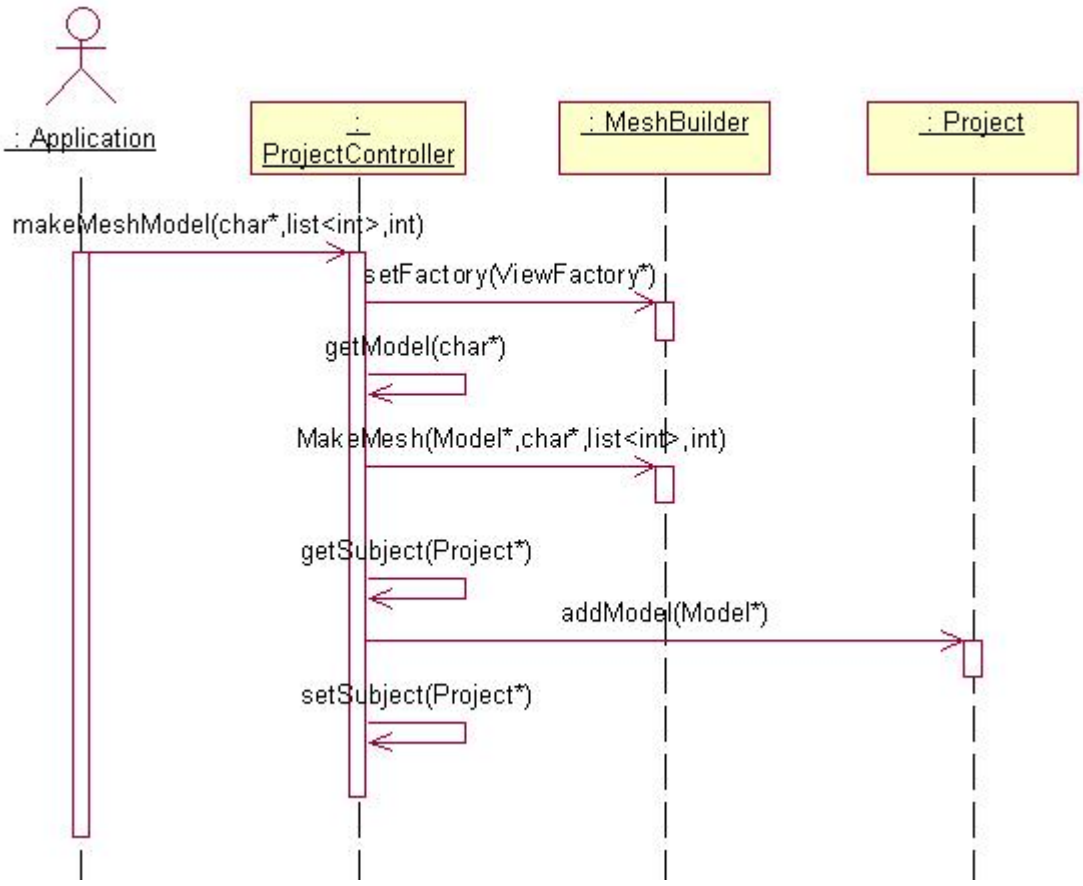

FIGURA 5.20 – Diagrama de seqüência: caso de uso *Make Mesh*

Apesar da implementação da classe *MeshBuilder* no núcleo do *framework*, para definir a construção de uma malha de elementos lineares de dois nós, é possível definir através de herança novas classes de construção de malhas nas aplicações, permitindo a especialização deste processo nas aplicações.

5.3.3 Casos de uso *New Material* e *New Section*

Através dos casos de uso *New Material* (FIGURA 5.21) e *New Section* (FIGURA 5.22) é possível definir novos materiais e novas seções e inseri-los no projeto para posterior aplicação aos elementos lineares de um modelo. Os objetos Material e Section são armazenados em um Projeto. Os elementos de cada modelo possuem apenas a identificação do material ou da seção transversal aplicada a eles. Para permitir a definição de materiais e de seções nas aplicações, o fluxo geral gerenciado pelo *frozen spot* aciona o objeto fábrica *Builder Factory* para a criação dos construtores de objetos apropriados. O objeto fábrica e os construtores são definidos nas aplicações através do *hot spot* de criação de objetos. O fluxo de definição dos objetos é então desenvolvido de forma especializada nas aplicações, sendo acionado pelo *frozen spot* que, após a criação dos objetos, retoma o controle da ação adicionando os objetos criados no modelo, procedendo as atualizações e os controles necessários para manter a consistência de dados.

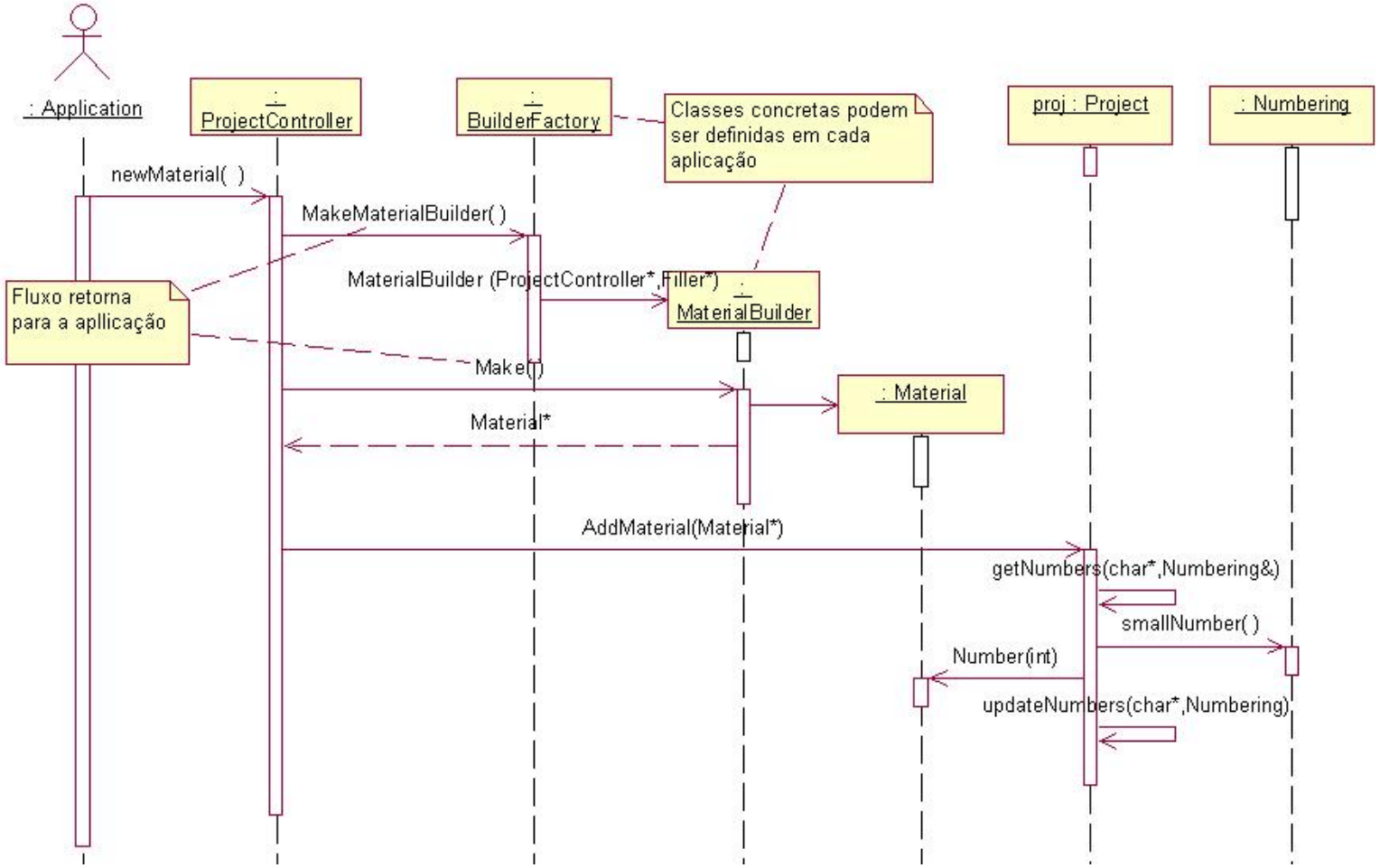

FIGURA 5.21 – Diagrama de seqüência: caso de uso *New Material*

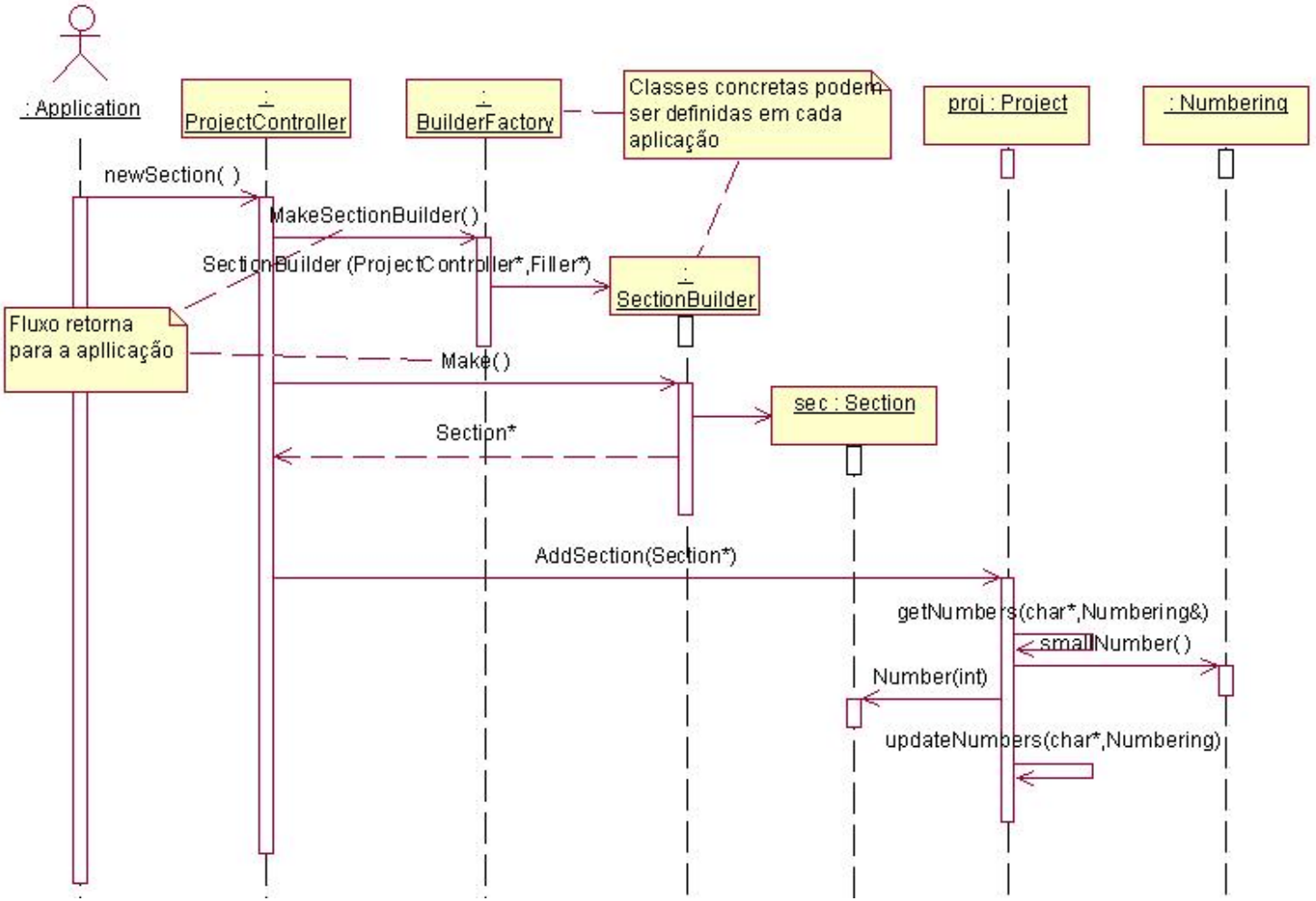

FIGURA 5.22 – Diagrama de seqüência: caso de uso *New Section*

5.3.4 Casos de uso *Apply Material* e *Apply Section*

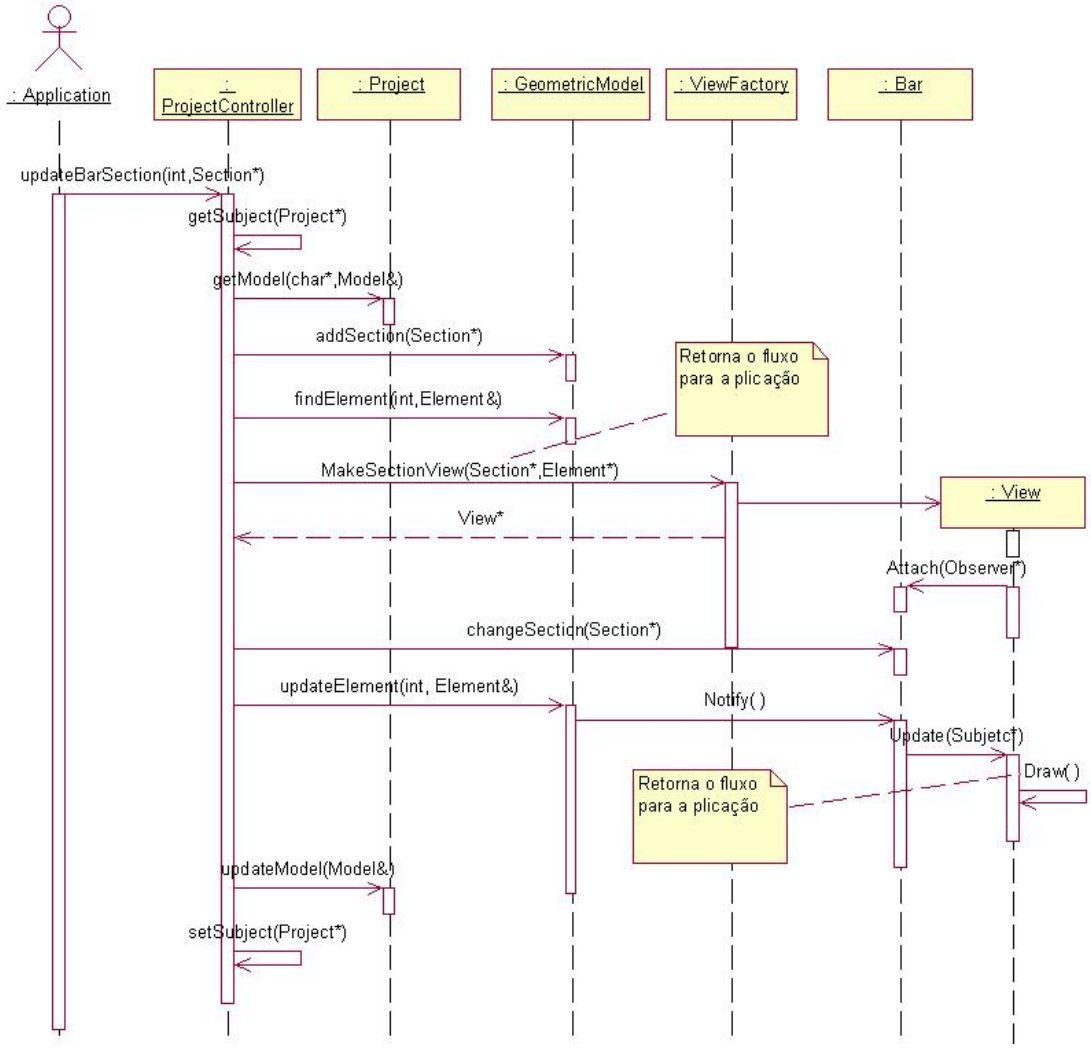

FIGURA 5.23 – Diagrama de seqüência: caso de uso *Apply Section*

Os casos de uso *Apply Material* (FIGURA 5.24) e *Apply Section* (FIGURA 5.22) resentam os fluxos de aplicação de materiais e de seções transversais a elementos lineares do modelo. Em ambos os casos de uso, praticamente todo o fluxo é gerenciado no *frozen spot* do *framework*. Apenas na criação de visualizações para as seções transversais no caso de uso *Apply Section,* e na atualização das representações gráficas dos elementos do modelo, o fluxo é repassado para as aplicações através da chamada à

função *updateElement*, que deve ser implementada nos *hot spots* de visualização, conforme já mencionado no item 4.2.1.

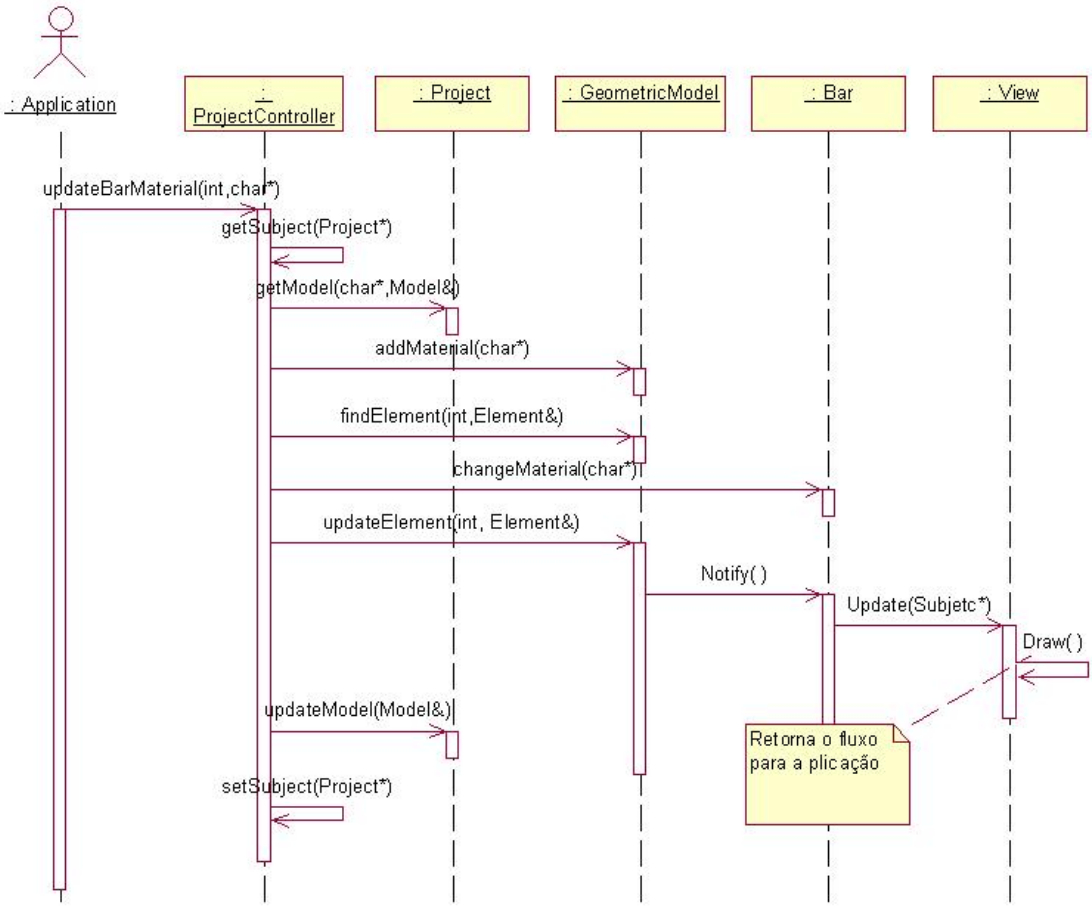

FIGURA 5.24 – Diagrama de seqüência: caso de uso *Apply Material*

Tanto materiais quanto seções são armazenados na classe de banco de dados *Project*. Para aplicar materiais e seções aos elementos do modelo, apenas as chaves de identificação dos objetos vinculados são inseridas nos elementos lineares e no modelo ao qual estes elementos estão vinculados. Através destas chaves os dados dos objetos vinculados são recuperados para consultas e atualizações. Os objetos do tipo *Material* não possuem uma representação gráfica, enquanto as seções transversais possuem uma representação gráfica independente da representação dos elementos lineares, permitindo tratamentos diferenciados para ambas as visualizações.

5.3.5 Caso de uso *Add Load*

O caso de uso *Add Load* (FIGURA 5.25) apresenta um fluxo mais complexo do que os casos de uso *Apply Material* e *Apply Section*, apresentados no item anterior. O diagrama de seqüência da FIGURA 5.25 apresenta o caso de uso através de um fluxo bastante simplificado, indicando apenas as principais chamadas realizadas pelo mesmo. Este caso de uso envolve vários objetos do modelo como: carregamentos, casos de carregamentos e métodos de carregamentos, e reúne em um mesmo fluxo a definição dos carregamentos e a aplicação dos mesmos aos elementos do modelo.

O *hot spot* de criação de elementos permite a definição do carregamento pelas aplicações através da definição de uma classe *Builder* para definição do processo de criação de uma carga. No *framework* foi definida a classe *LoadBuilder* que realiza um tipo de inserção genérico que pode ser utilizado pelas aplicações. Assim como nos casos anteriores, a visualização do carregamento é definida através do *hot spot* de visualização. Todo o restante do fluxo de adição do carregamento ao modelo, como definição de um caso de carregamento, definição de um método para este caso, vinculação entre carregamento e este caso, etc., é definido no *frozen spot* do *framework*.

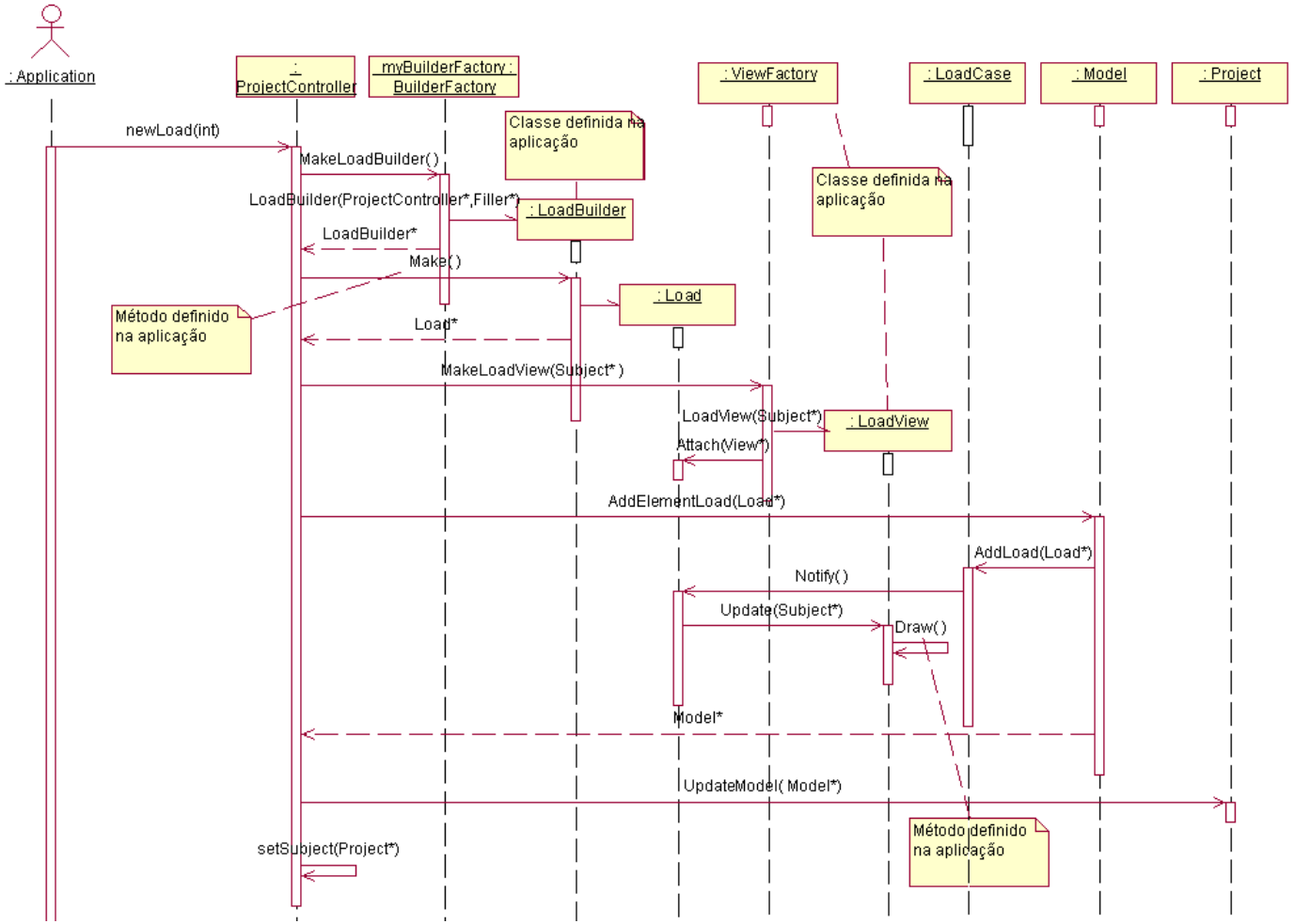

FIGURA 5.25 – Diagrama de seqüência: caso de uso *Add Load*

5.3.6 Caso de uso *Add Restraints*

O caso de uso *Add Restraint* (FIGURA 5.26) permite a aplicação de restrições nodais a nós existentes no modelo. As restrições nodais são definidas por um objeto do tipo *Values* definido pelo componente *Math Classes,* parametrizado por inteiros. Estes valores não são aplicados diretamente aos nós, mas são armazenados em um dicionário da classe model que vincula o número de um nó restrito às restrições definidas. Este dicionário evita a definição de um atributo extra em todos os nós do modelo enquanto apenas uma parte deles acaba recebendo algum tipo de restrição. Da mesma forma, a busca por nós restritos é efetuada de uma forma mais rápida e direta através deste dicionário, sem necessitar fazer a varredura de todo o modelo para encontrar os nós com restrições.

A definição da visualização adotada para as restrições nodais fica a cargo das classes de visualização definidas nas aplicações e o fluxo é repassado para estas tanto na criação dos objetos de visualização quanto na atualização das representações gráficas destes objetos (*hot spot* de visualização).

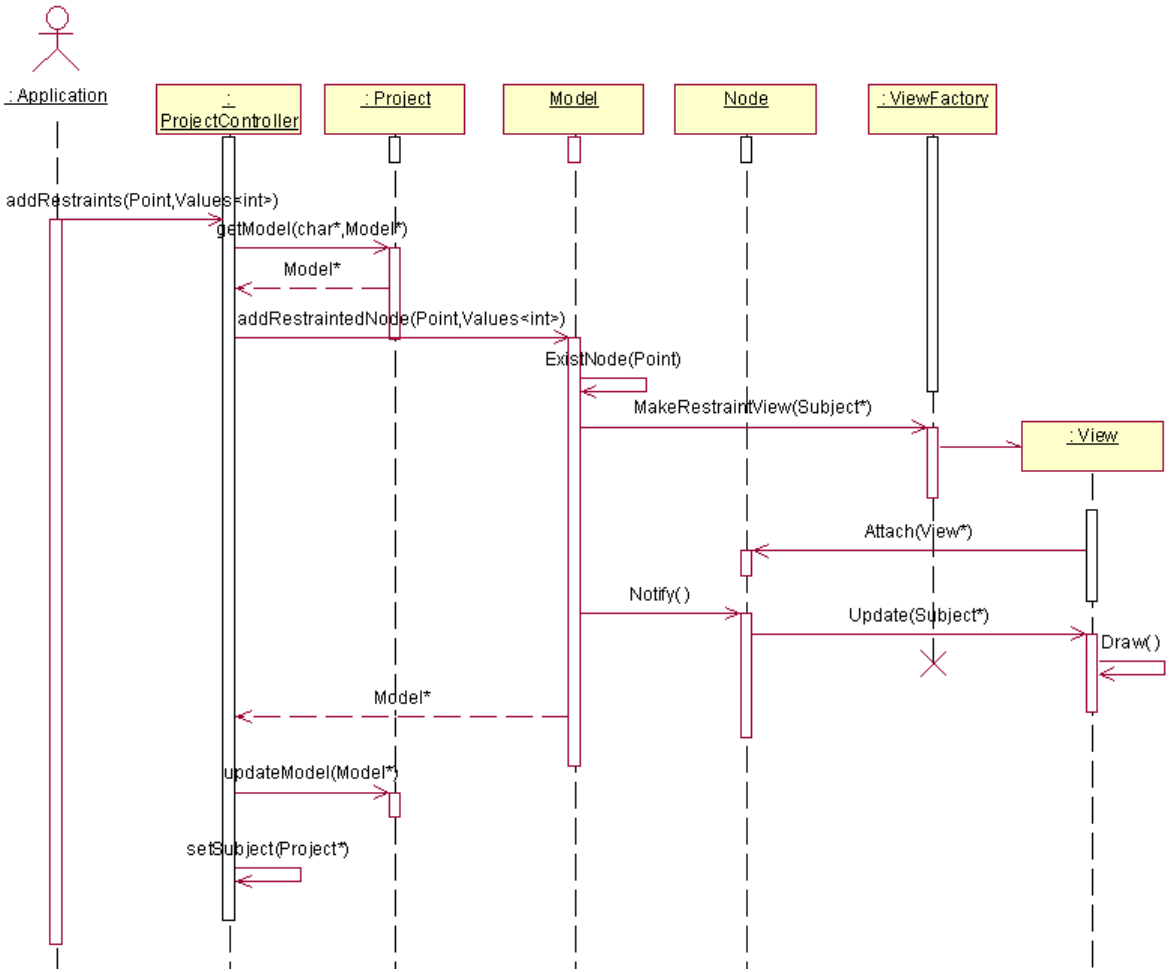

FIGURA 5.26 – Diagrama de seqüência: caso de uso *Add Restraint*

5.3.7 Casos de uso *Load Project* e *Save Project*

Os casos de uso *Load Project* e *Save Project* implementam a persistência dos dados do modelo. A leitura e a gravação de todos os dados armazenados em um projeto é realizada através da chamada aos métodos *FillIn* e *FillOut,* definidos nas classes derivadas da classe *Component*. Cada classe é responsável por ler ou gravar seus atributos e quando um atributo é um objeto de uma classe do tipo *Component,* sua função de leitura ou de gravação é chamada pela classe possuidora.

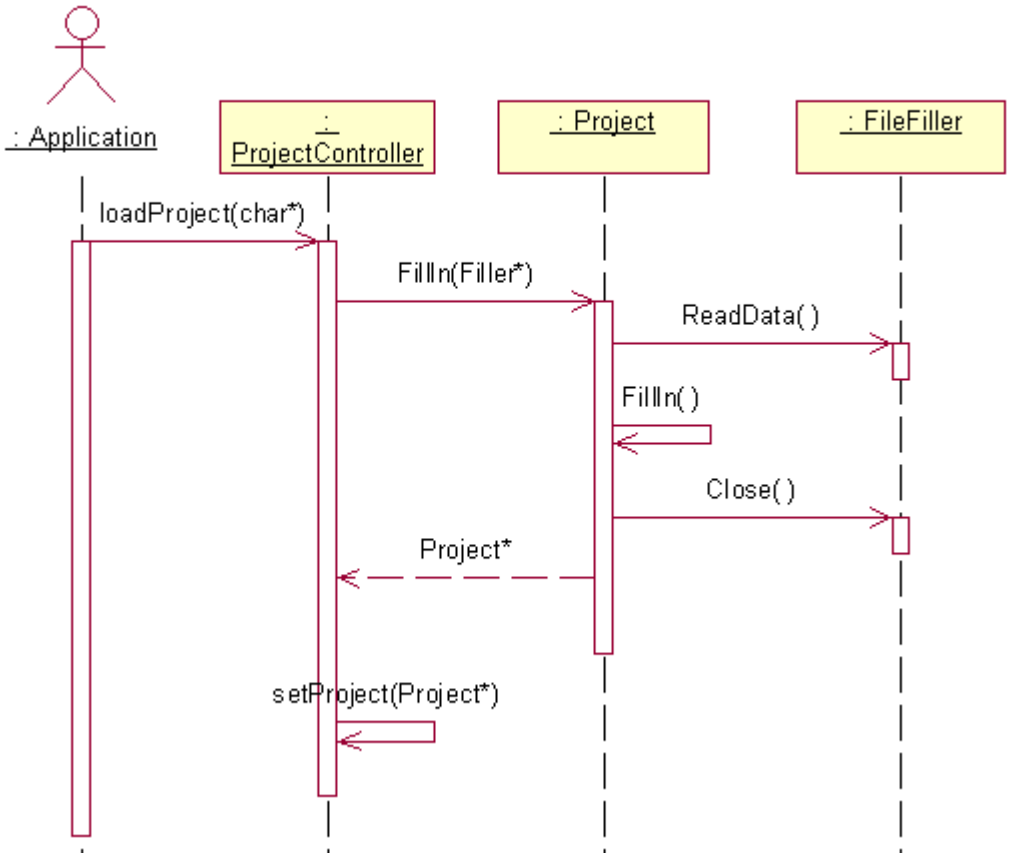

FIGURA 5.27 – Diagrama de seqüência: caso de uso *Load Project*

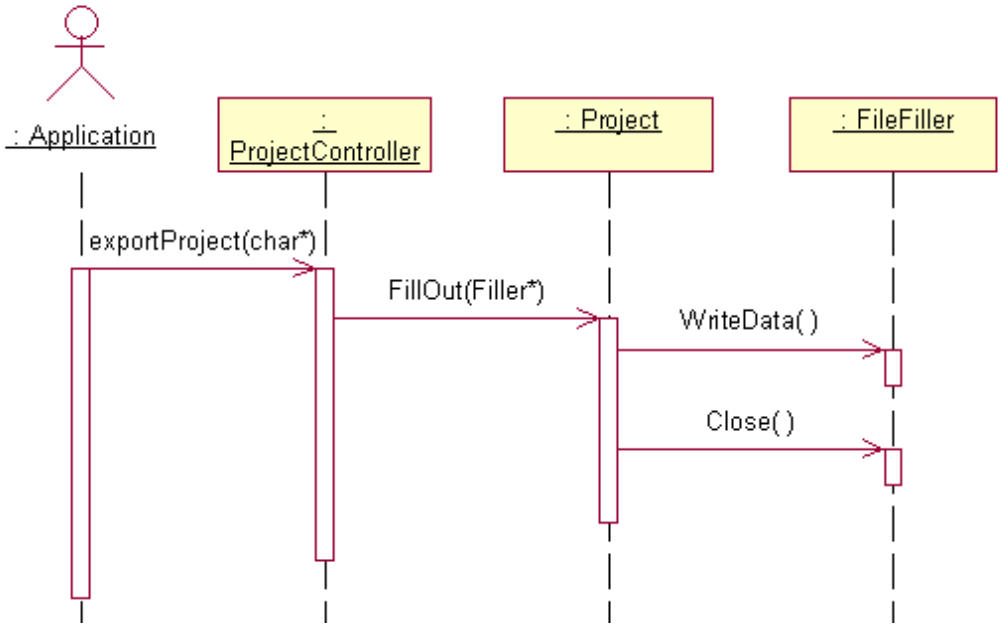

FIGURA 5.28 - – Diagrama de seqüência: caso de uso *Save Project*

O tipo de arquivo de persistência é definido pelas classes *Filler* utilizadas. A classe *FileFiller* implementa a gravação e leitura de todos os tipos de dados presentes na aplicação em arquivos ASCII. A classe *FileFiller* e a chamada recursiva das funções *FillIn* e *FillOut* pelas classes que formam um projeto, definem o formato padrão adotado pelo *framework* para garantir a persistência de dados em sua versão inicial. Nesta versão os arquivos de persistência apresentam informações sobre os elementos de cada modelo criado dentro de um projeto. As informações dos elementos são agrupadas por blocos de tipos de objetos. São informados dados sobre seções e materiais do projeto, e para cada modelo são indicadas as seções e os materiais nele aplicados. Para os nós são informados: coordenadas, as cargas inseridas e barras vinculadas. As cargas são agrupadas em um bloco com informações sobre caso de carregamento ao qual cada carga está vinculada, valores, direção de aplicação e tipo de carga. Para as barras são informados: numeração, incidência nodal, eixos locais, seção, material e cargas aplicadas e o caso de carregamento de cada carga aplicada. As informações sobre restrições ou liberações e valores prescritos são agrupados por modelo, com indicação de valores e de elementos de aplicação.

Para futuras aplicações pretende-se vincular o *framework* a um banco de dados para permitir, através de uma organização relacional de dados, uma otimização do armazenamento de dados e da busca por informações.

5.3.8 Casos de uso *Export Data* e *Import Data*

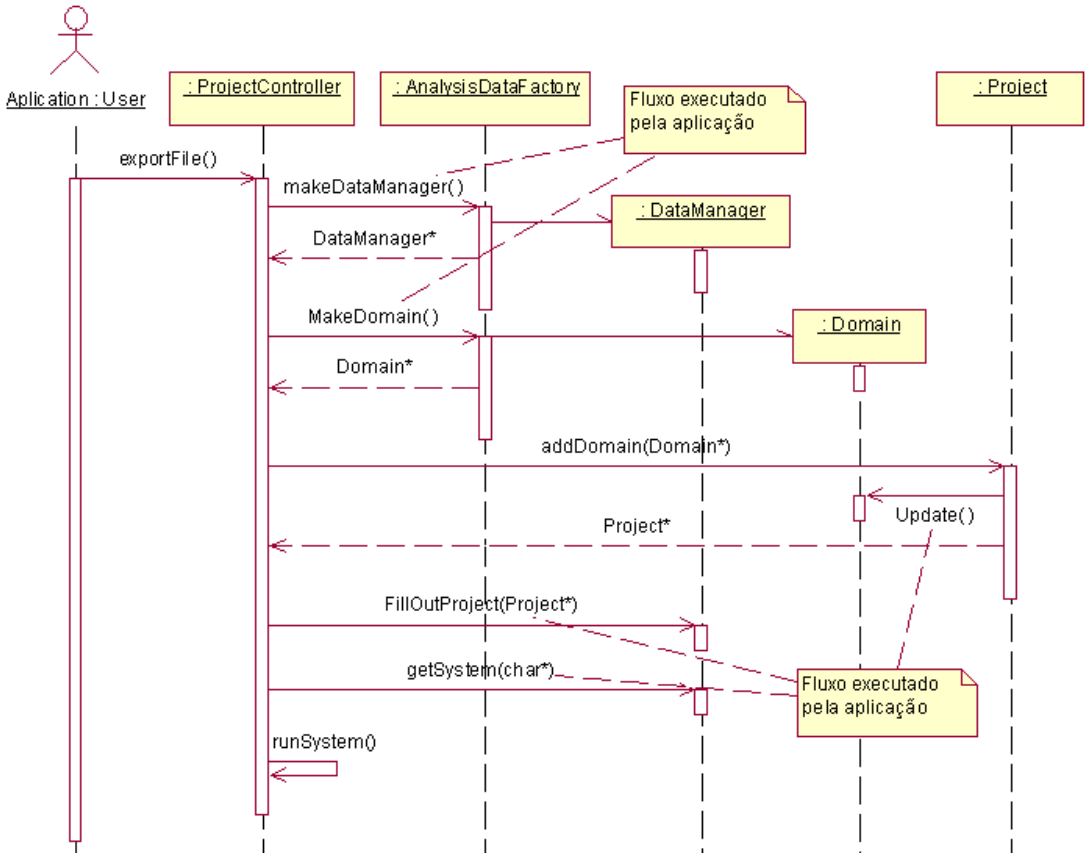

FIGURA 5.29 – Diagrama de seqüência: Caso de Uso *ExportData*

Estes casos de uso são responsáveis por implementar o fluxo de operações para transferência de dados entre o modelo gerenciado pelo *framework* e os sistemas externos utilizados nas diversas etapas do processo.

A classe *ProjectController* utiliza uma classe fábrica para obter as instâncias dos objetos do tipo *DataManager* e *Domain* especializados para cada aplicação. Estes

objetos são responsáveis por executar a formatação dos dados de transferência e de domínio nas aplicações.

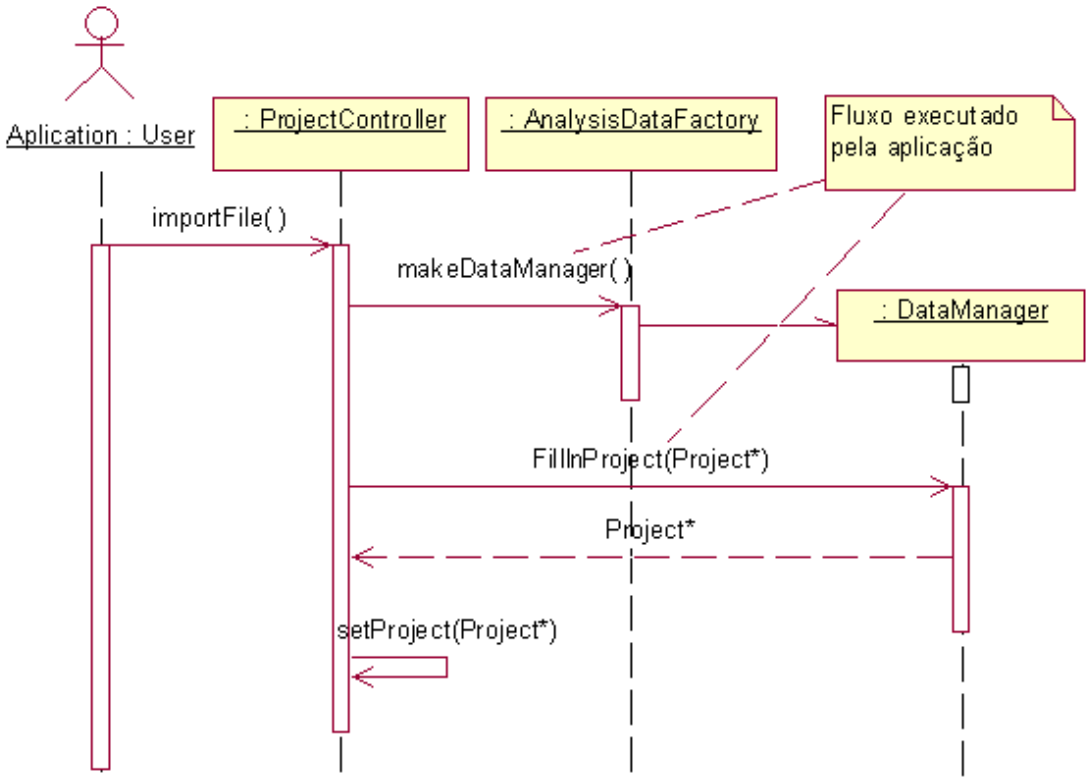

FIGURA 5.30 - Diagrama de seqüência: Caso de Uso *ImportData*

No caso de uso *Export Data* (FIGURA 5.29), antes de acionar o sistema externo para executar a análise da estrutura, a classe de controle *ProjectController* repassa à classe *Project* um objeto de domínio. Ao adicionar este objeto a seu conjunto de dados a classe *Project* solicita ao mesmo que se atualize. O método *Update,* implementado nas classes do tipo *Domain,* utiliza as classes implementadas através do *hot spot Filler*, para obter os dados de domínio a serem repassados para o sistema de análise. Com os dados de domínio já definidos, o método *FillOutProject* da classe *DataManager* é acionado pelo controlador. Através deste método, implementado nas classes derivadas de *DataManager*, os dados do modelo são formatados e gravados nos arquivos de transferência de dados. Após o preenchimento destes arquivos o objeto de controle

obtém a identificação do sistema a ser executado do objeto *DataManager*, e procede à chamada deste sistema.

5.3.9 Caso de uso *Draw Model*

Como pode existir mais de um modelo de dados em um projeto, e como cada modelo possui dados específicos, o usuário pode necessitar atualizar a representação gráfica para que a mesma mostre o modelo que ele deseja alterar, ou consultar, ou visualizar resultados. Para ativar a visualização de um modelo específico foi definido o caso de uso Draw Model. Este caso de uso pode ser acionado diretamente pelo usuário ou ser acionado por outro caso de uso que necessita editar um determinado modelo, fazendo com que o modelo a ser editado seja apresentado na tela gráfica.

Uma outra utilização deste caso de uso é a vinculação dos objetos de dados às suas visualizações durante a reconstrução de um modelo através de um documento de persistência. Como apenas as classes de dados são persistidas de uma seção para outra, através dos arquivos ASCII, quando um projeto é carregado pelo caso de uso *Load Project*, todas os objetos de visualização devem ser criados e as vinculações entre modelo e visualização têm de ser realizadas. Estas vinculações são executadas pelo caso de uso *Draw Model*, após o preenchimento dos dados de todos os objetos de um projeto pelo caso de uso *Load Project*.

Este caso de uso obtém cada objeto de dados de um modelo (classe *Model*) do projeto. Se o objeto de dados ainda não possui uma visualização vinculada a ele, a classe fábrica derivada de *ViewFactory* é acionada retornando o objeto de visualização especificado para o elemento. O controlador realiza a vinculação entre modelo e visualização e atualiza seu objeto de dados, o que aciona o mecanismo de propagação de mudanças, fazendo com que a visualização do elemento seja construída e adicionada à tela gráfica. Após todas as vinculações necessárias entre modelo e visualização, o objeto do tipo *Model* é reinserido no projeto.

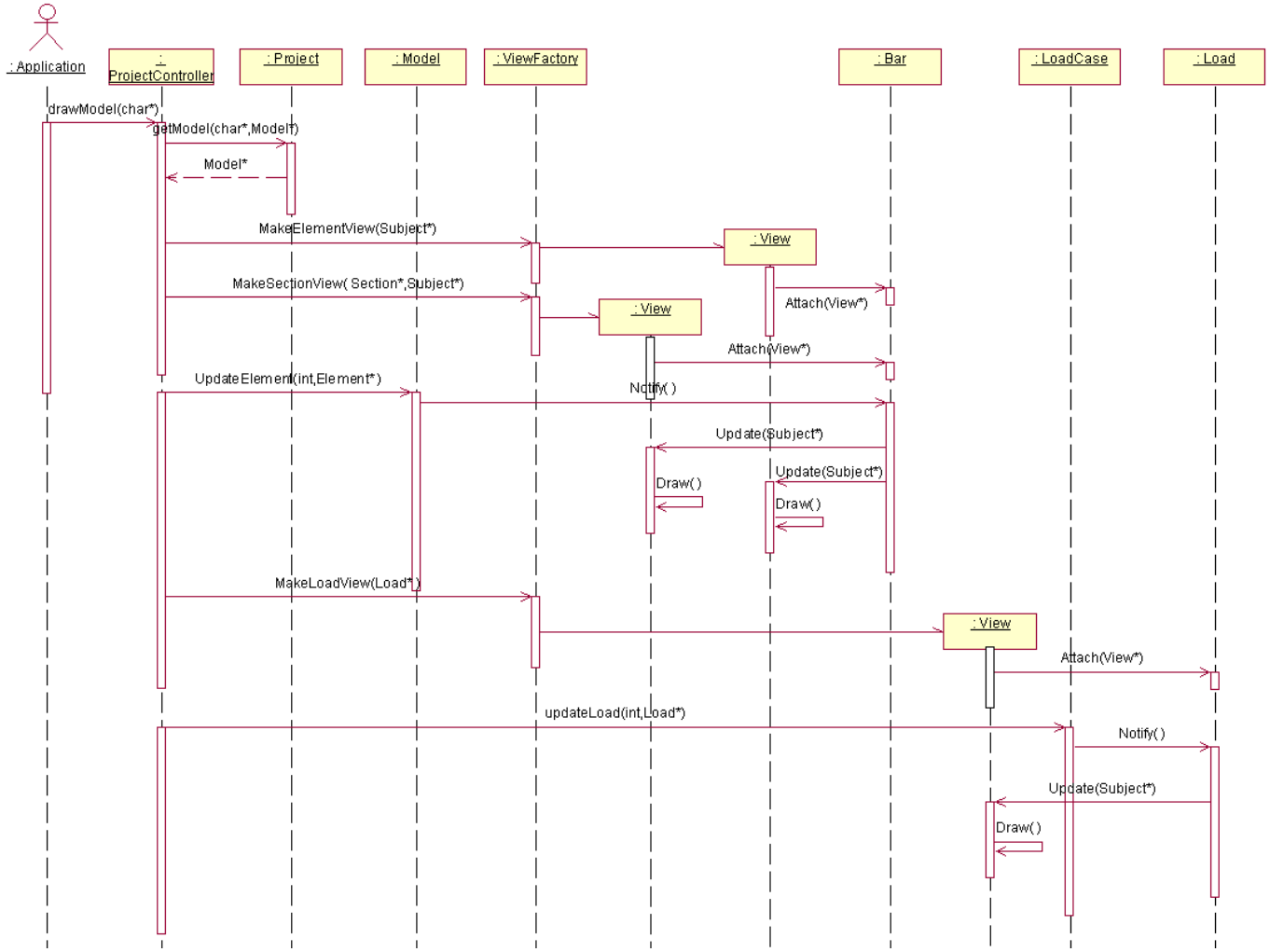

FIGURA 5.31 - Diagrama de seqüência: Caso de Uso *DrawModel*

5.3.10 Casos de uso *Read Displacements* e *Read Results*

De uma maneira semelhante à implementada no caso de uso *Draw Model*, os casos de uso *Read Displacements* e *Read Results*, permitem que dados obtidos dos sistemas externos sejam inseridos ou atualizados em um modelo (classe *Model*), atualizando as visualizações dos elementos, ou criando as visualizações para os dados obtidos, caso estas ainda não existam.

A leitura dos dados resultantes da análise é realizada de forma independente para deslocamentos e para esforços, sempre através das classes *DataManager*. Assim, nestes casos de uso são acionados três *hot spots*: *DataManager*, *ViewFactory* e *View*. Através destes *hot spots* o fluxo é retornado para a aplicação para leitura de dados, criação dos objetos de visualização e criação das representações inseridas na tela gráfica.

Para permitir a representação da deformada da estrutura e dos esforços em cada etapa de processos incrementais, os objetos de visualização não são gravados junto com os dados de modelo. Estes objetos são criados a cada chamada dos casos de uso, evitando atualizações constantes do modelo, e utilização excessiva de memória para armazenagem de objetos que não são utilizados durante todo processo de projeto, mas apenas na avaliação visual dos resultados da análise.

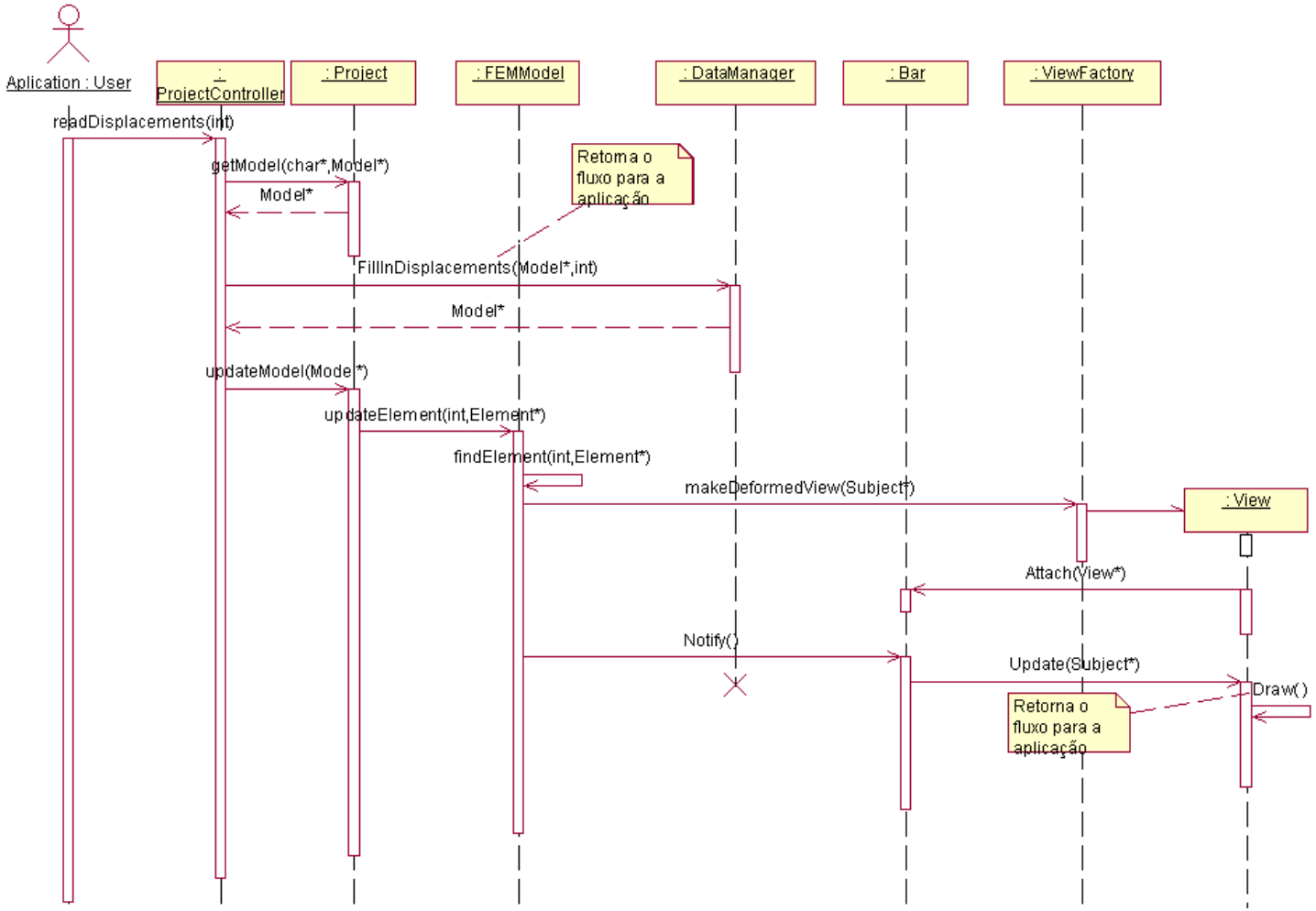

FIGURA 5.32 – Diagrama de seqüência: Caso de uso *Read Displacements*

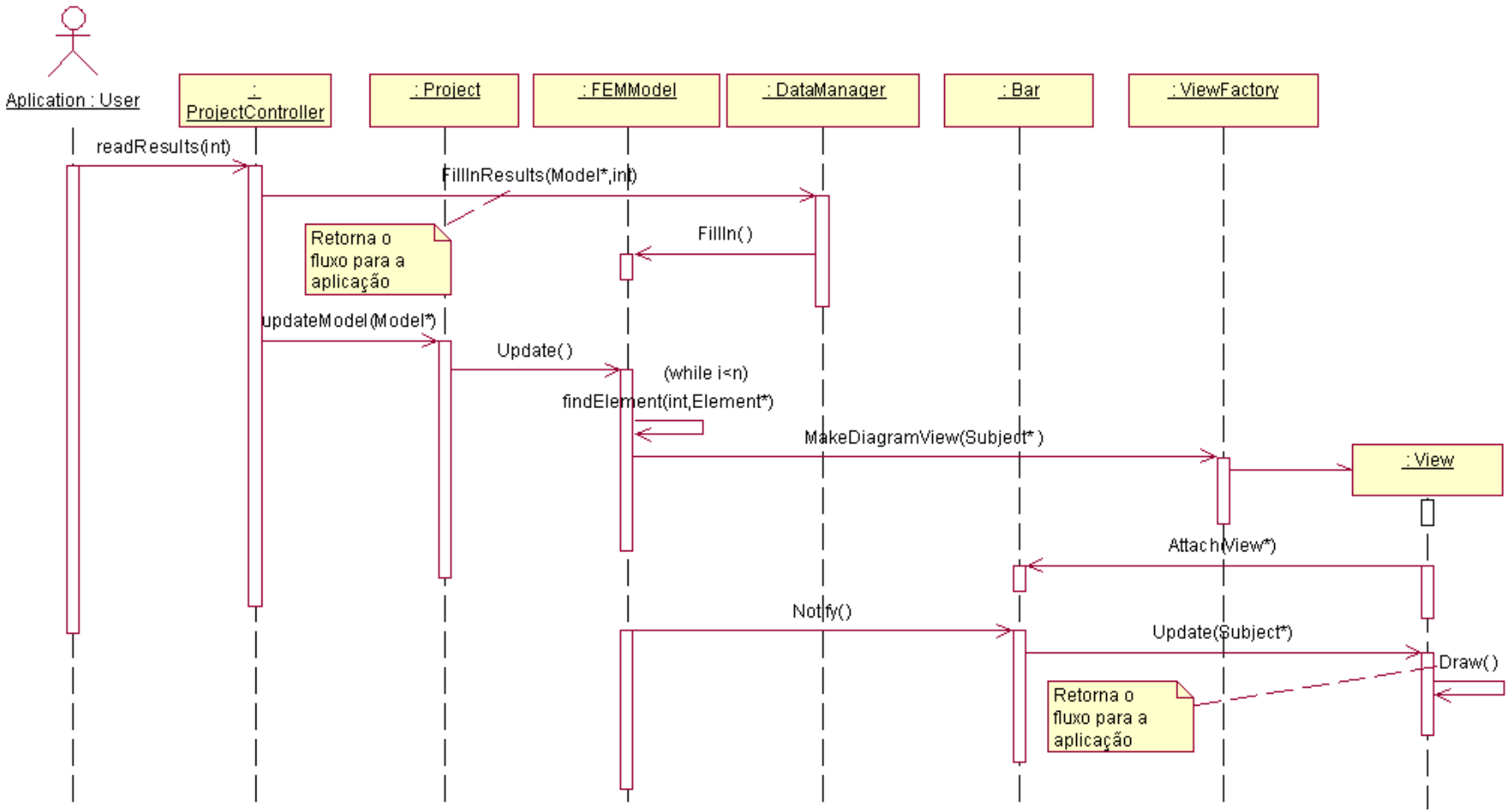

FIGURA 5.33 – Diagrama de seqüência: caso de uso *Read Results*

6

APLICAÇÕES E VALIDAÇÕES

O presente capítulo apresenta as aplicações desenvolvidas como parte desta tese para validação do *framework* apresentado no capítulo 5. Inicialmente é realizada uma descrição geral das aplicações. Em seguida são apresentados a estrutura de cada aplicação, com indicação das implementações realizadas e um ou mais estudos de caso que exemplificam o funcionamento das mesmas.

6.1 Descrição geral das aplicações

Foram implementados três sistemas CAD, sendo uma plataforma genérica semelhante ao Modelador Gráfico 3D apresentado por SOARES (2006) e dois sistemas específicos: um para estruturas de concreto pré-moldado (novo PREMOLD) e um para estruturas metálicas (STEELMOLD). Foram ainda implementados cinco sistemas de integração: um para geração automática do desenho de torres de transmissão (*TowerSystem*), um para integração com um sistema de análise não linear geométrica de treliças (*NLG Analysis*), um para para integração com um sistema de análise não linear de pórticos planos (*PLANLEP Analysis*), um sistema de integração entre o sistema de análise não linear de pórticos planos e um sistema de verificação da resistência de perfis metálicos em temperatura ambiente e em situação de incêndio VeriTemp.

O primeiro sistema CAD desenvolvido foi nomeado *GeometricModeler*. Nele foi implementada a funcionalidade básica do Modelador Gráfico 3D (SOARES, 2006), à qual foram acrescentadas algumas outras funcionalidades como geração de modelos refinados a partir do modelo geométrico e definição de seções geométricas compostas.

Os outros dois sistemas CAD foram desenvolvidos a partir do *GeometricModeler*: um para estruturas de concreto pré-moldado e outro para estruturas metálicas prediais. Estes sistemas implementam automações para o lançamento de elementos estruturais, semelhante à modelagem realizada pelo PREMOLD, porém com a implementação da automação de leitura e edição de bibliotecas de seções.

Os sistemas de integração foram desenvolvidos utilizando os modeladores criados nas aplicações citadas acima, fazendo a integração destes com sistemas externos de análise ou de dimensionamento. Para permitir que o sistema final seja utilizado a partir de um único executável, estes sistemas foram desenvolvidos como aplicações gerenciadas pela plataforma gráfica, mas não dependentes desta. Para qualquer das aplicações de integração apresentadas é possível apresentá-la como um sistema autônomo, que aciona o sistema CAD apenas para a visualização e edição de dados,

executando a transferência destes para os demais sistemas de forma completamente independente da plataforma.

6.2 Aplicação: *Geometric Modeler*

A primeira aplicação desenvolvida a partir do *framework* REMFrame foi um sistema gráfico para automação da modelagem de estruturas reticuladas. Esta modelagem é realizada através da definição do modelo unifilar formado por barras e nós. Após a definição deste modelo são permitidos: inserção de seções transversais, definição de materiais e aplicação de carregamentos.

Além de permitir o lançamento da estrutura reticulada, neste sistema CAD foram implementadas a leitura e a gravação dos dados gerenciados pelo sistema em arquivos ASCII. A persistência dos dados de um projeto foi garantida, nesta etapa de desenvolvimento, através destes arquivos. Esta forma de persistência foi inicialmente adotada também como instrumento de teste de aplicação das ferramentas de transferência e de formatação de dados, previstas para o *framework*.

Para viabilizar o modelador gráfico foi utilizada a plataforma gráfica comercial de arquitetura aberta utilizada nos demais trabalhos desenvolvidos pelo grupo CADTEC. No desenvolvimento deste sistema foi necessário instanciar classes de visualização para cada elemento do modelo, derivadas da classe *View* do componente *Classes MVC*, além de implementar os mecanismos de entrada de dados através da classe *ARXFiller,* derivada da classe *Filler,* e de caixas de diálogo.

O sistema CAD desenvolvido disponibiliza uma representação gráfica genérica para uma estrutura reticular, passível de ser utilizado em outros sistemas desenvolvidos através do *framework* e que utilizem a mesma plataforma gráfica. Este sistema permitiu testar e avaliar a aplicabilidade real do *framework* como base para o desenvolvimento

de outros sistemas CAD. O custo (tempo e número de implementações) demandado para o desenvolvimento somente desta aplicação, desconsiderando-se o custo de desenvolvimento do *framework*, foi consideravelmente menor do que os custos de desenvolvimento dos demais sistemas do grupo CADTEC. Os procedimentos de controle de dados implementados são mais simples que os controles necessários para manter a consistência dos sistemas originais. A estrutura de cada classe de visualização definida no sistema CAD é significativamente menor e mais simples que as estruturas das classes dos sistemas originais.

É necessário esclarecer, porém, que algumas decisões de projeto do *Geometric Modeler* diferem significativamente da filosofia adotada na definição da funcionalidade dos primeiros sistemas CAD desenvolvidos. Para os sistemas CAD anteriores, o principal requisito de desenvolvimento é permitir ao usuário trabalhar sobre os elementos das aplicações como se estes fossem entidades próprias da plataforma gráfica. Ou seja, estes sistemas priorizam a aplicação dos comandos de edição da plataforma sobre os elementos modelados. Para esta aplicação definiu-se como requisito principal garantir a aplicação do *framework* no desenvolvimento da mesma, ou seja, deve-se garantir que as alterações nos dados do modelo sejam sempre realizadas através das classes de controle. Desta forma, não se permite que os comandos de edição da plataforma sejam aplicados diretamente sobre as representações gráficas. As edições dos elementos devem ser realizadas através de comandos próprios, desenvolvidos pela aplicação, que repassam as solicitações dos usuários às classes de controle. As visualizações são então atualizadas, representando o novo estado dos elementos após as alterações solicitadas sobre os mesmos.

Nos sistemas CAD anteriores, não havia uma separação entre dados, controle de fluxos e representação. Assim, uma mesma entidade é responsável por receber solicitações, tratar seus dados mediante as solicitações realizadas e atualizar sua representação. Ao permitir a aplicação dos comandos de edição da plataforma diretamente sobre os objetos da aplicação, o desenvolvimento destes sistemas exige a validação de inúmeros tipos de alterações possíveis sobre as entidades, o que torna o controle de tais sistemas complexo e sujeitos a resultados inesperados, pela combinação destes comandos de edição sobre uma entidade.

Na aplicação *Geometric Modeler*, todas as solicitações do usuário são recebidas pelas classes de controle e transmitidas às classes de modelo. As classes de visualização têm a responsabilidade apenas de atualizar a sua representação sempre que o modelo informar uma alteração de estado. Esta restrição ao uso dos comandos de edição da plataforma sobre as entidades gráficas evita que alterações sejam realizadas sobre a visualização sem corresponder a uma alteração nos dados gerenciados pelo modelo. Assim, é possível realizar alterações sobre as entidades, porém estas alterações são processadas através de comandos próprios da aplicação, e não através dos comandos da plataforma.

Desta forma, o esforço computacional necessário para o desenvolvimento deste módulo gráfico foi definido tanto pela utilização do *framework* como base de desenvolvimento, quanto pela forma de edição permitida para as entidades do sistema. Uma vez que todo o gerenciamento de dados foi implementado pelo núcleo do *framework*, o desenvolvimento do módulo gráfico foi realizado pela definição dos comandos para inserção e edição de dados no modelo e pela definição dos elementos gráficos a serem utilizados na visualização dos objetos.

O *Geometric Modeler* foi organizado em dois componentes: *ARXModeler* e *DBXModeler* (FIGURA 6.1). O componente *ARXModeler* é responsável pelo controle de fluxos e de dados, enquanto no componente *DBXModeler* são definidas as classes de visualização a partir da DLL de representação gráfica da plataforma. Estes componentes são definidos como DLLs que são carregadas e gerenciadas pela plataforma.

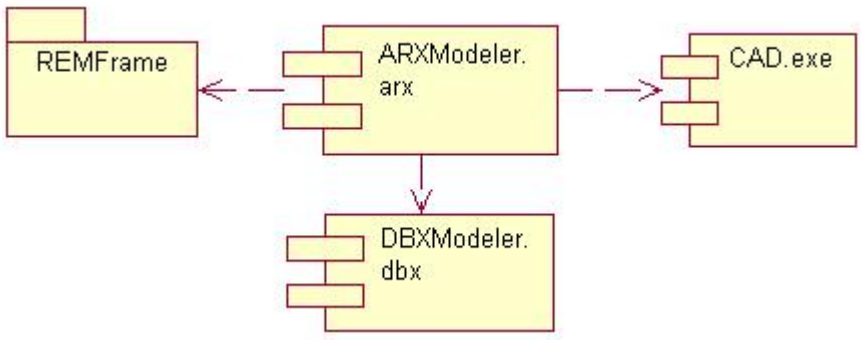

FIGURA 6.1 – Componentes da aplicação *Geometric Modeler*

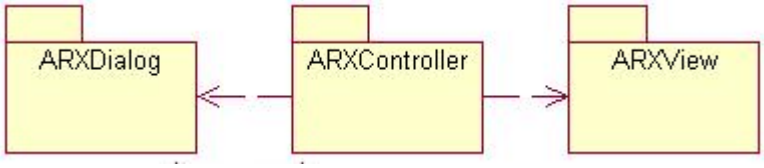

FIGURA 6.2 – Pacotes do componente ARXModeler

6.2.1 O componente ARXModeler

O componente *ARXModeler* foi organizado em três pacotes: um pacote de controle de fluxos e de criação de objetos, um pacote de visualização e um pacote de interface de usuário (FIGURA 6.2). No pacote *ARXController* são instanciados alguns *hot spots* do *framework* para definição de comandos, controle de fluxos de dados, leitura e gravação de dados, criação de objetos de visualização (FIGURA 6.3). No pacote de interface de usuário são definidas classes para implementação de caixas de diálogo para alguns comandos de criação e de edição de elementos implementados (FIGURA 6.4). As classes definidas neste pacote utilizam também classes definidas pelo *framework* REMFrame para a construção dos objetos criados. O pacote de visualização reúne as classes derivadas da classe *View* do *framework* REMFrame (FIGURA 6.5).

ARXController

No pacote *ARXController* (FIGURA 6.3) foram definidas as classes *ARXProjectController*, *ARXCommands*, *ARXFiller*, *ARXViewFactory*, *SectionFiller* e *MaterialFiller*.

A plataforma gráfica permite o trabalho simultâneo com múltiplos documentos. Para responder a esta característica cada documento possui um par, classe de controle

(*ARXProjectController*) e classe de banco de dados (*Project*), acoplado. A classe de controle fica responsável por gerenciar a criação dos objetos seção, material e carregamento, específicos para o sistema, e por controlar os fluxos de dados e de solicitações entre usuário e modelo, e o fluxo de dados entre sistemas externos e modelo. A classe de controle ao ser criada, recebe como parâmetro um objeto da classe fábrica *ARXViewFactory*. Esta classe fica responsável por instanciar os objetos de visualização definidos no pacote *ARXView* através de seus métodos fabrica (padrão *Abstract Factory*). A classe de controle faz, então, a vinculação entre cada objeto de modelo criado e inserido no banco de dados e sua visualização. Desta forma temos uma única tríade controle, modelo e visualização por documento.

A classe *ARXCommands* implementa os comandos de criação e edição de entidades. Devido a características específicas da plataforma gráfica, foi criada uma classe de comando, e os comandos implementados no sistema gráfico foram definidos como métodos desta classe. Estes métodos definem a obtenção de dados do usuário e acionam os métodos da classe *ARXProjectController*.

A classe *ARXProjectController* é derivada da classe de controle *ProjectController* do *framework*. Em sua implementação são redefinidos os métodos acionados nos casos de uso para a criação de seções, de materiais e de carregamentos, vinculando as classes do pacote *Builder* à classe de leitura de dados *ARXFiller*. Esta última é responsável por definir os métodos de leitura e escrita de dados utilizados no sistema na formatação utilizada pela plataforma gráfica. Na classe *ARXProjectController* são ainda implementados alguns métodos chamados pelo núcleo do *framework*, mas que dependem da inserção de dados pelo usuário. Estes métodos são utilizados por outros do núcleo do *framework* para processar as informações obtidas e realizar suas tarefas, de modo independente da aplicação implementada.

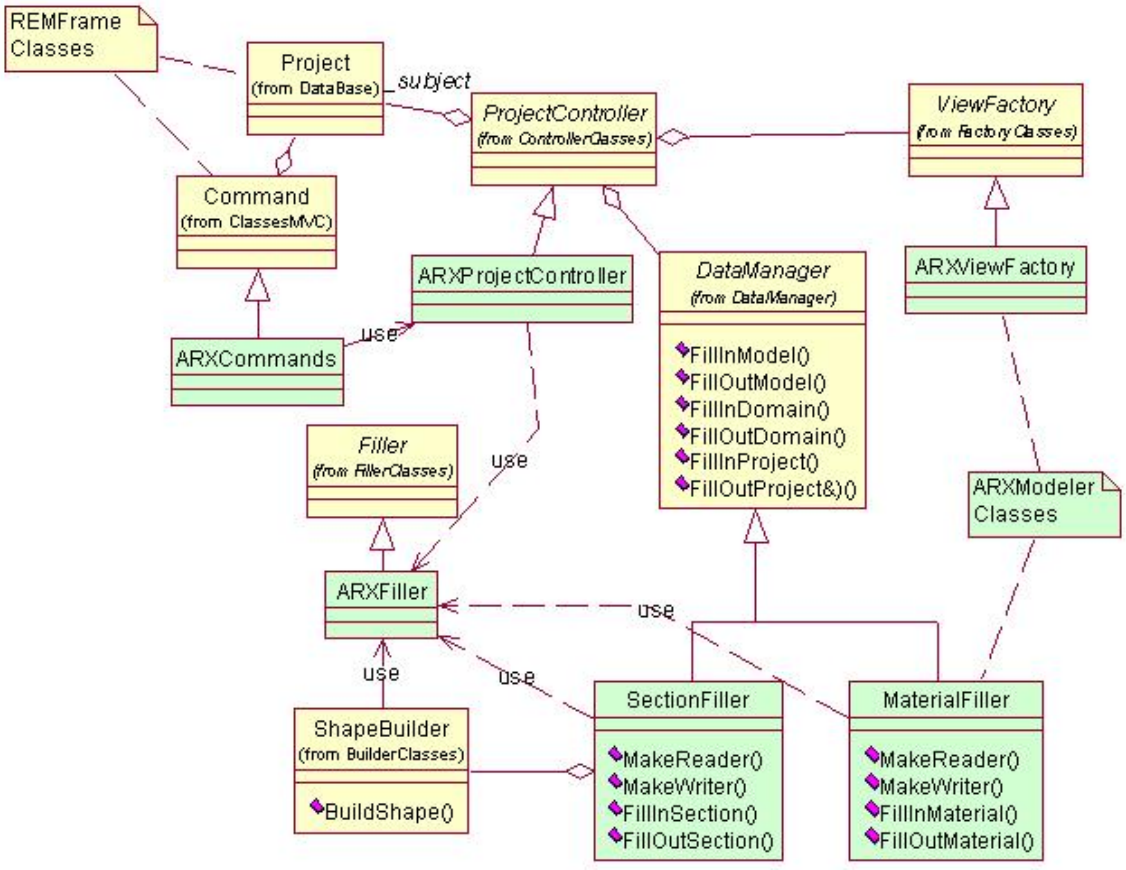

FIGURA 6.3 – Classes do pacote ARXController

As classes *SectionFiller* e *MaterialFiller*, derivadas da classe *DataManager*, foram implementadas para fazer a leitura e gravação, em arquivos ASCII, de objetos do tipo *Section* e *Material* respectivamente, na formatação padrão definida pelo *framework*. Estes arquivos são utilizados para a persistência dos dados do modelo. Através destes arquivos é possível definir um "banco de dados de seções e de materiais", permitindo carregar em um projeto seções e materiais definidos em outro projeto.

ARXDialog

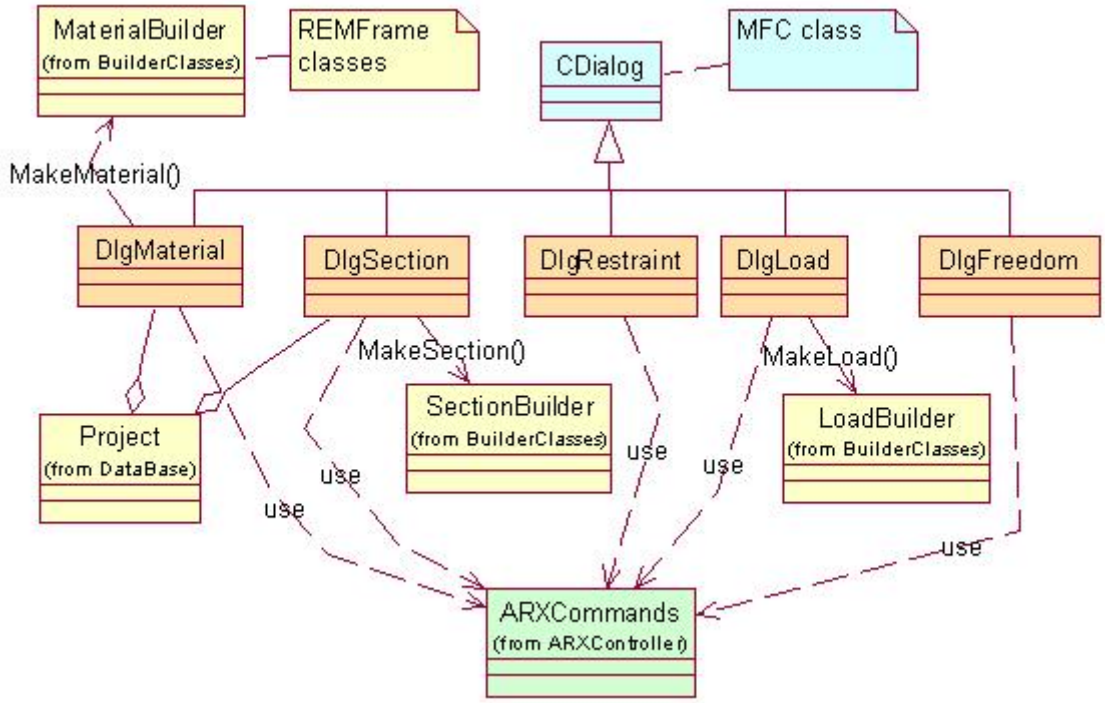

FIGURA 6.4 – Classes do pacote ARXDialog

O pacote *ARXDialog* (FIGURA 6.4) engloba as classes derivadas da classe *CDialog* do *framework* MFC. Foram definidas classes para a utilização de caixas de diálogo nos comandos de criação, inserção e aplicação de materiais e de seções no modelo, aplicação de restrições nodais e liberações de elementos. Para a criação de seções, de materiais e de carregamentos são utilizadas as classes *Builder* do *framework* REMFrame. As seções e os materiais podem ser aplicados a várias barras do modelo e por isso as classes de caixas de diálogo para estes objetos têm uma vinculação com a classe *Project* acessando o projeto de cada documento ativo.

ARXView

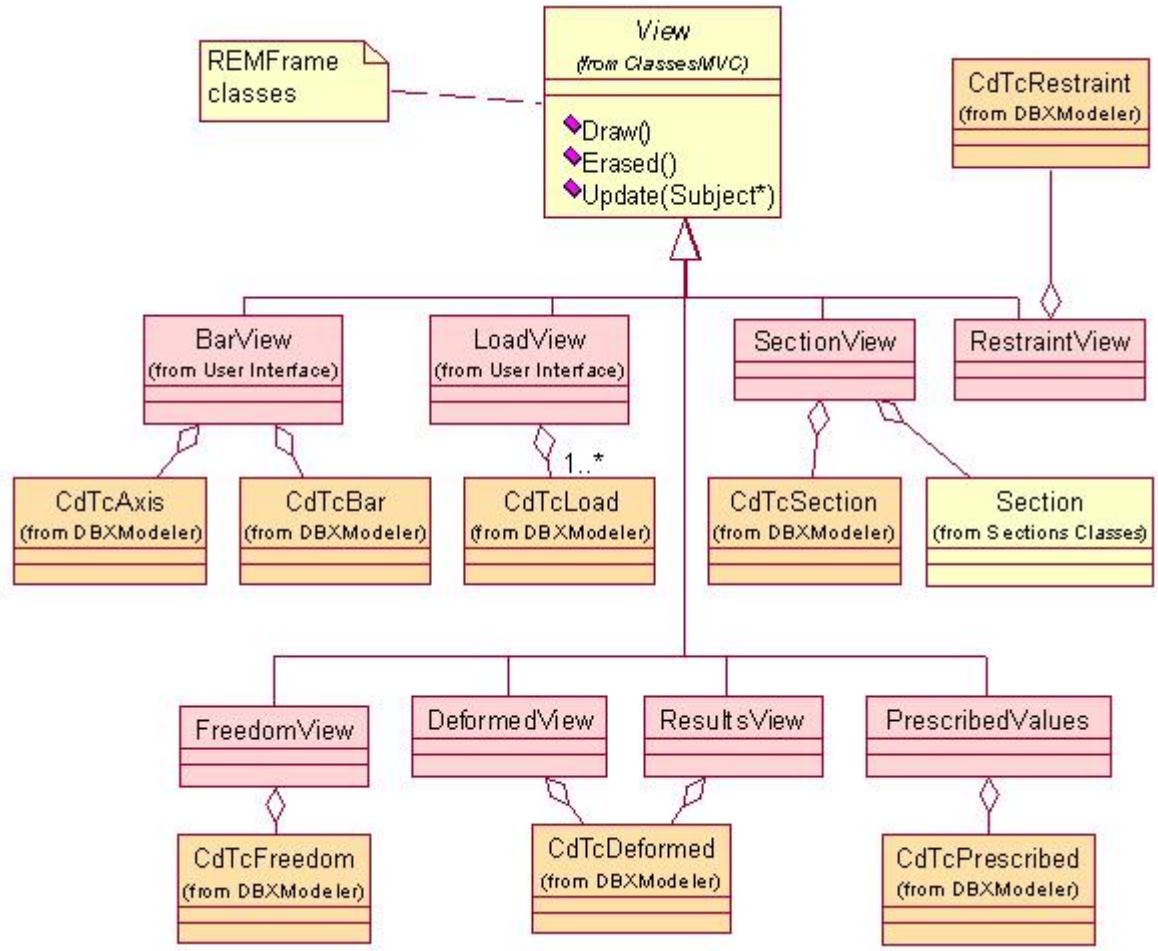

FIGURA 6.5 – Classes do pacote ARXView

No pacote *ARXView* (FIGURA 6.5) foram definidas classes de visualização, derivadas da classe *View,* gerando a representação gráfica de barras e nós, de carregamentos, de restrições nodais, de seções transversais, de deformadas da estrutura, etc. Nestas classes foram implementados os métodos *Draw* e *Erased,* definidos na interface da classe *View*.

Na definição das classes de visualização foi adotado o padrão *Adapter* (item 4.2.5). Através deste padrão a funcionalidade gráfica da plataforma é utilizada pelas classes derivadas de *View*. Objetos das classes implementadas através da DLL de representação gráfica da plataforma foram utilizados como atributos das classes de

visualização, e a construção das representações gráficas é realizada através destes objetos (FIGURA 6.5). As funções *Draw* e *Erased*, quando acionadas, fazem o tratamento necessário de dados, repassando-os para os objetos da plataforma. O método *Draw* faz então a inserção destes objetos no banco de dados da plataforma, enquanto o método *Erased* exclui estes objetos de tal banco.

6.2.2 Componente DBXModeler

Para o desenvolvimento de aplicações que utilizam a plataforma AutoCAD, as classes definidas através do uso da DLL de representação gráfica da plataforma devem ser agrupadas em um componente próprio que recebe a denominação DBX[24]. Através do padrão *Adapter* as classes de visualização, derivadas da classe *View,* e definidas no componente *ARXModeler,* utilizam a funcionalidade gráfica das entidades da plataforma para criar a visualização dos elementos do modelo. Não é permitido às classes do componente DBX manter qualquer vinculação a dados dependentes da plataforma. Depois de inseridos no banco de dados da plataforma os objetos gráficos passam a ser gerenciados pela mesma. Assim, para manter a coerência entre visualização e modelo, as classes do componente DBX recebem, além dos dados a serem representados, um atributo que é o identificador do elemento do modelo que o objeto gráfico representa. Através destes identificadores, torna-se possível permitir ao usuário indicar, através dos objetos de visualização, os elementos de modelo que deseja

[24] DBX: database extension.

editar. A classe de controle transmite então as solicitações do usuário para o objeto de modelo representado pela visualização selecionada. Após a realização das alterações no modelo, este solicita à visualização que se atualize. As classes do componente *ARXModeler* armazenam os identificadores da visualização criada e inserida no banco de dados da plataforma. Através dos identificadores armazenados nas classes derivadas de *View,* os objetos gráficos da plataforma são recuperados e atualizados com os novos dados resultantes do processo desencadeado.

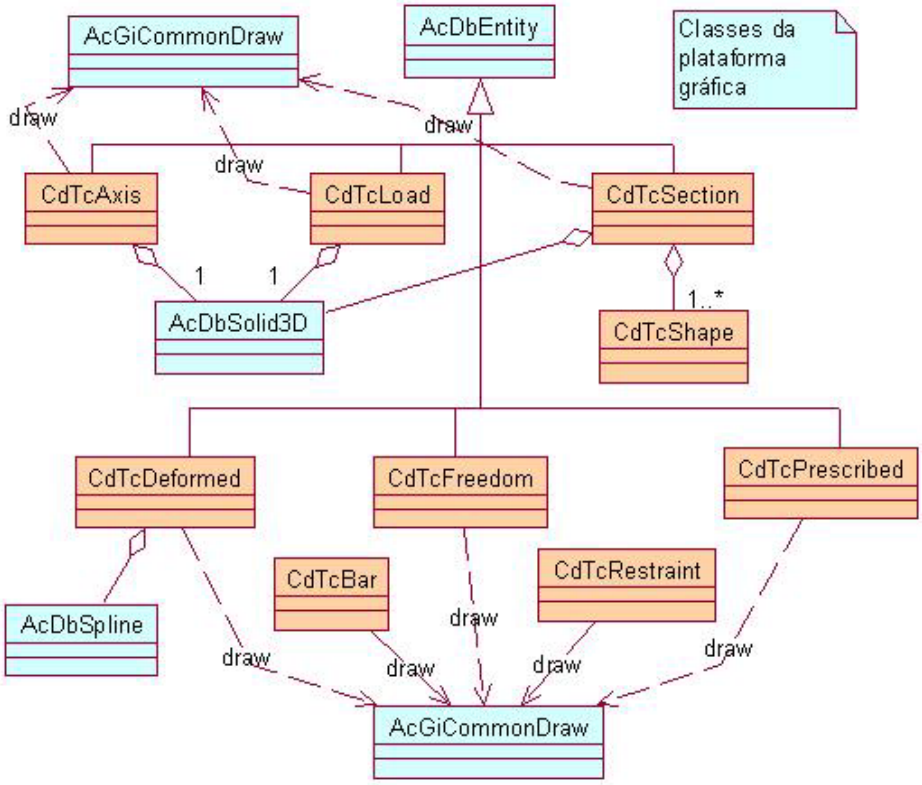

FIGURA 6.6 – Classes do componente DBXModeler

Para a realização de representação gráfica, cada classe definida utiliza as primitivas de desenho da interface gráfica (classe *AcGiCommonDraw*) da plataforma. Para a representação da geometria sólida são criados objetos sólidos gerenciados pelo banco de dados da plataforma (classe *AcDbSolid3D*).

6.2.3 Estudo de Caso

Neste item são apresentadas duas caixas de diálogo para os comandos de definição de seções transversais (FIGURA 6.7) e de materiais (FIGURA 6.8) para os elementos do modelo através do modelador *Geometric Modeler*. São mostradas ainda duas imagens obtidas de modelos inseridos com o uso deste modelador, com inserção de cargas (FIGURA 6.9) e de seções transversais (FIGURA 6.10). Os exemplos dos demais sistemas desenvolvidos utilizam sempre como base o modelador *Geometric Modeler* e, conseqüentemente, os exemplos apresentados também demonstram o funcionamento desse modelador.

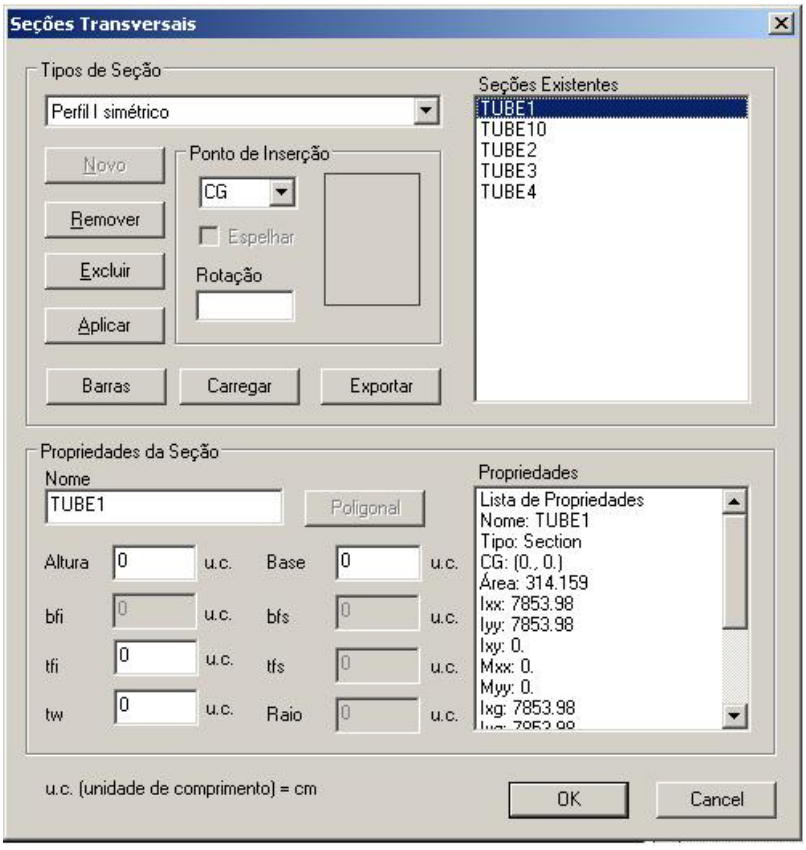

FIGURA 6.7 – Caixa de diálogo para definição, edição e aplicação de seções transversais.

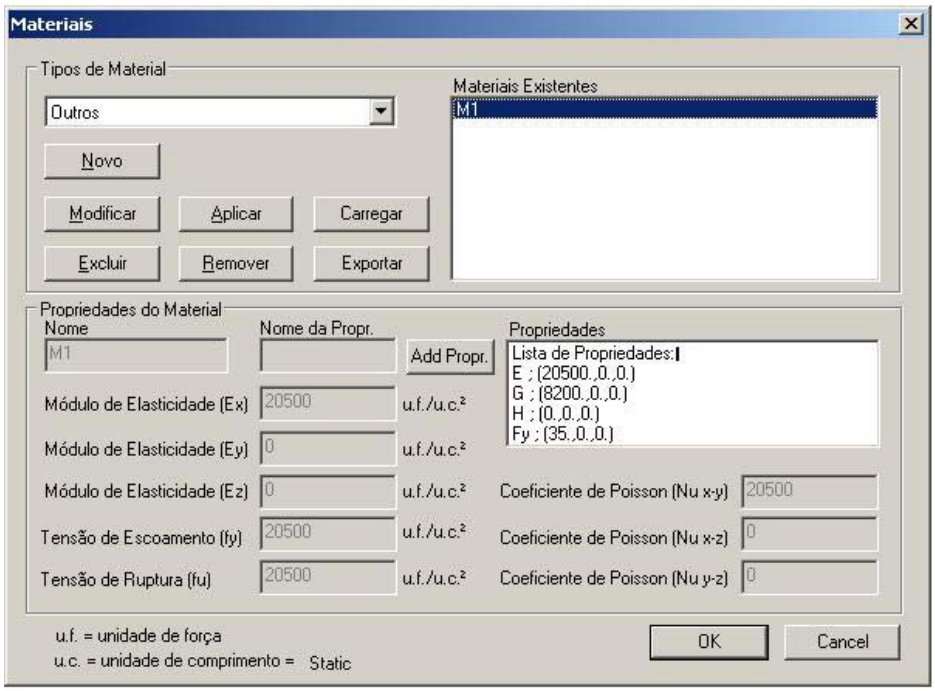

FIGURA 6.8 – Caixa de diálogo para definição, importação e aplicação de materiais.

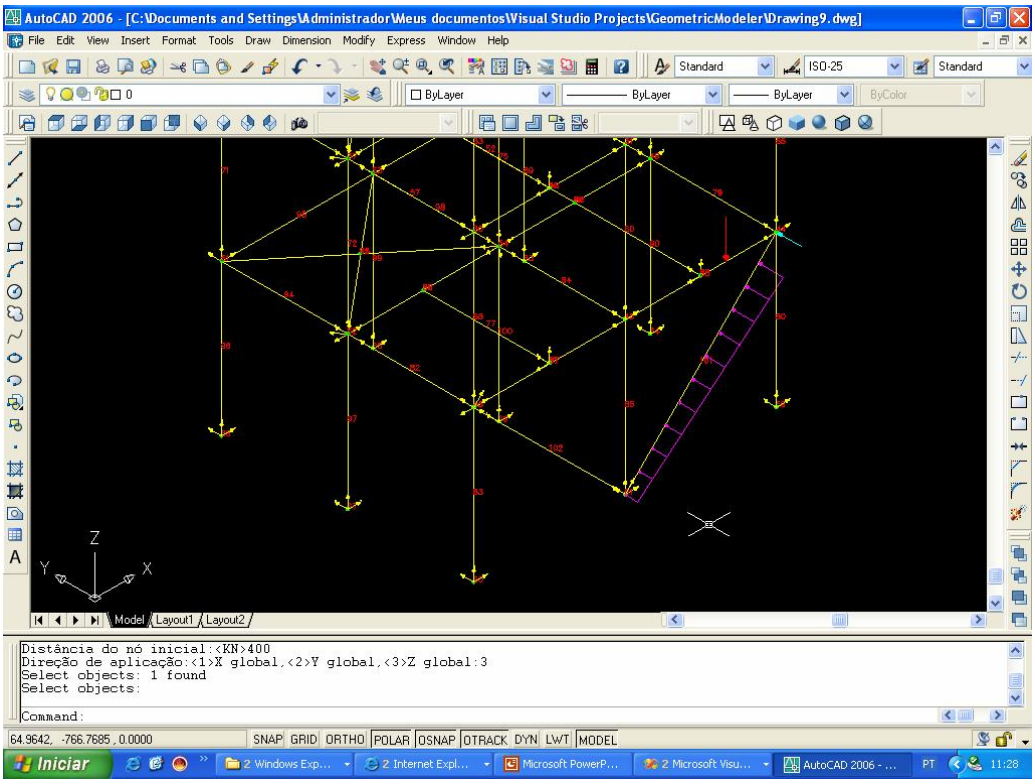

FIGURA 6.9 – Modelo unifilar com aplicação de carregamentos.

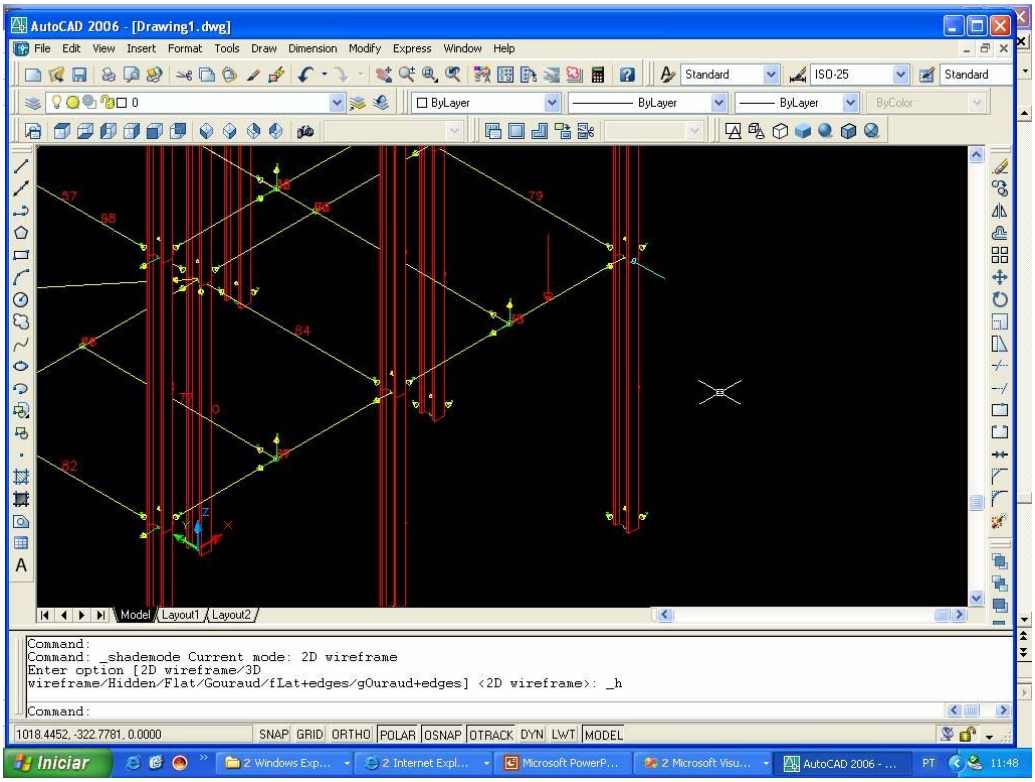

FIGURA 6.10 – Modelo com inserção de seções transversiais e de carregamentos.

6.3 Aplicação: TowerSystem

A segunda aplicação desenvolvida realiza a integração do sistema gráfico *Geometric Modeler* com o sistema de análise e dimensionamento de torres de transmissão abordado pelo TowerCAD (MAGALHÃES, 2002), (FIGURA 6.11).

O desenvolvimento desta aplicação visou tanto validar os recursos desenvolvidos para a integração de sistemas externos através do *framework* quanto validar a funcionalidade do modelador desenvolvido.

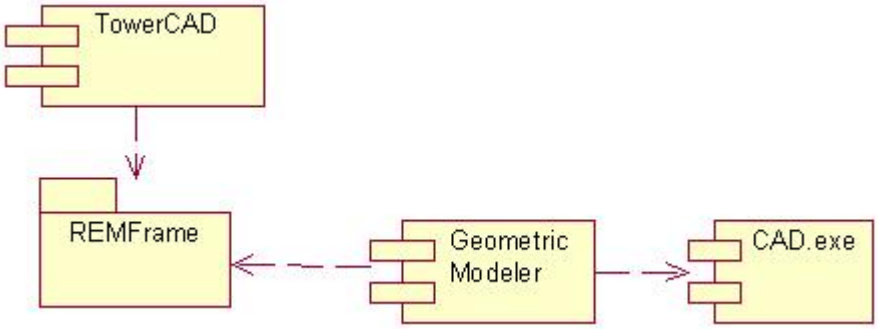

FIGURA 6.11 – Componentes da aplicação TowerSystem

Para a implementação deste sistema foi necessário definir duas novas classes *TSFactory*, uma classe fábrica, e *Tower_System,* uma nova classe de gerenciamento de dados derivada da classe *DataManager*. Esta classe realiza a leitura de dados segundo a formatação utilizada por MAGALHÃES (2002) e cria um modelo de uma estrutura de torre a partir dos dados obtidos, utilizando o modelador *Geometric Modeler* para representar o modelo na plataforma CAD. A FIGURA 6.12 apresenta as classes definidas na aplicação *TowerSystem*. A classe *Tower_System* é criada através da classe *TSFactory*, uma classe fábrica do tipo *AnalysisDataFactory.*

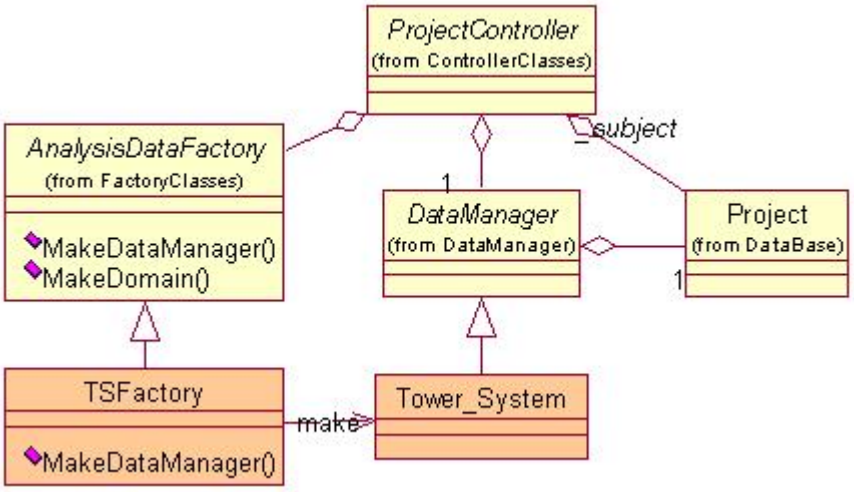

FIGURA 6.12 – Classe da aplicação TowerSystem

Após a criação do modelo da torre, o qual é gerenciado pelo núcleo do *framework*, todos os comandos do modelador podem ser aplicados sobre o mesmo, permitindo alterações na geometria, nas seções transversais, etc. A inserção dos stubs, ou de qualquer outra barra na estrutura pode ser realizada pela criação de um novo elemento, inserção do mesmo no modelo e aplicação de uma seção transversal. A ligação entre os elementos não foi tratada nesta versão da aplicação.

6.3.1 Estudo de Caso

As FIGURA 6.13 a FIGURA 6.15 apresentam uma estrutura de torre de transmissão cujos modelos foram gerados automaticamente pelo *TowerSystem* a partir da leitura dos arquivo de dados apresentados por MAGALHÃES (2002). Salientamos que a versão atual dessa aplicação não realiza o tratamento de ligações e que a finalidade do desenvolvimento da mesma foi a implementação dos mecanismos de importação de dados segundo um formato pré-definido pelo sistema externo. O sistema torres de transmissão foi escolhido como primeiro sistema de integração para permitir uma base de comparação com os mesmos modelo geométricos gerados pelo sistema TowerCAD, possibilitando avaliar a validação do modelo importado pelas classes do *framework* o qual passa a permitir o gerenciamento os dados da estrutura ao longo do processo.

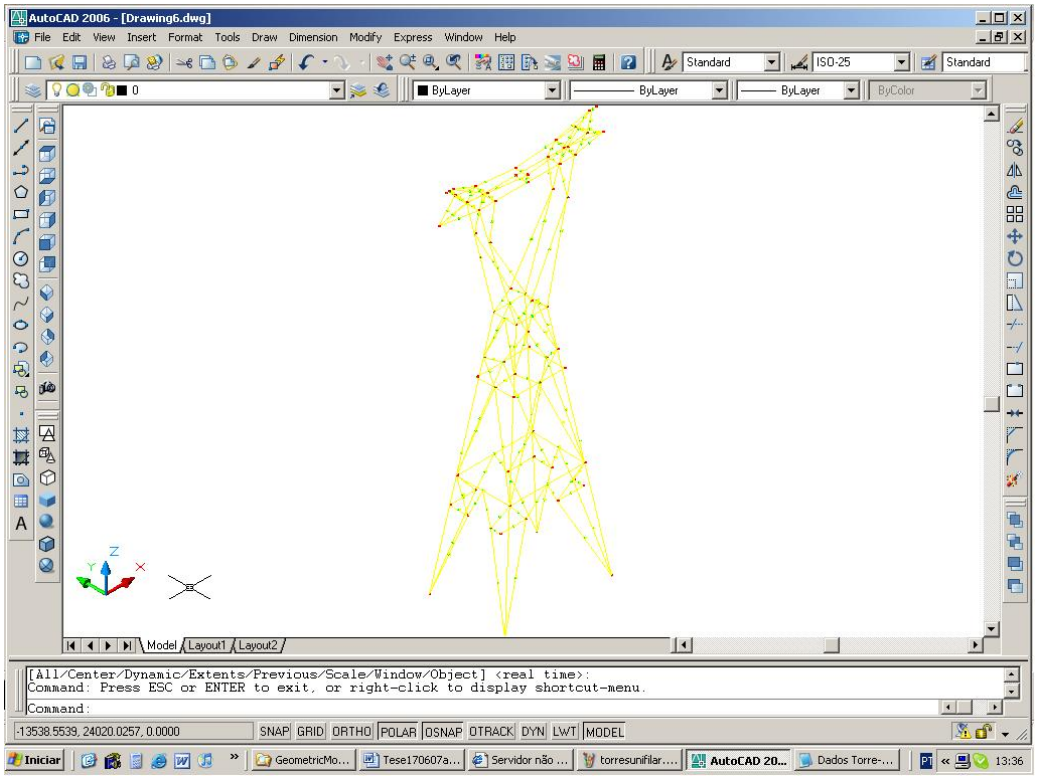

FIGURA 6.13 – Modelo unifilar de torre de transmissão gerado a partir do *TowerSystem*.

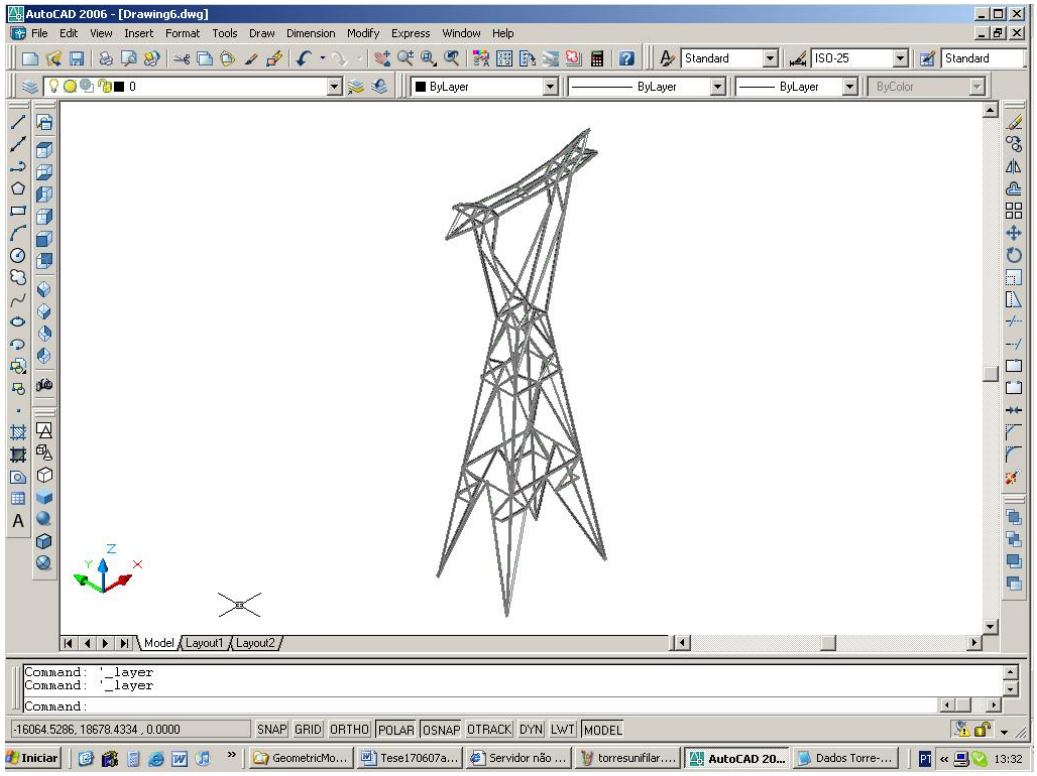

FIGURA 6.14 – Modelo sólido de torre de transmissão gerado a partir do *TowerSystem*.

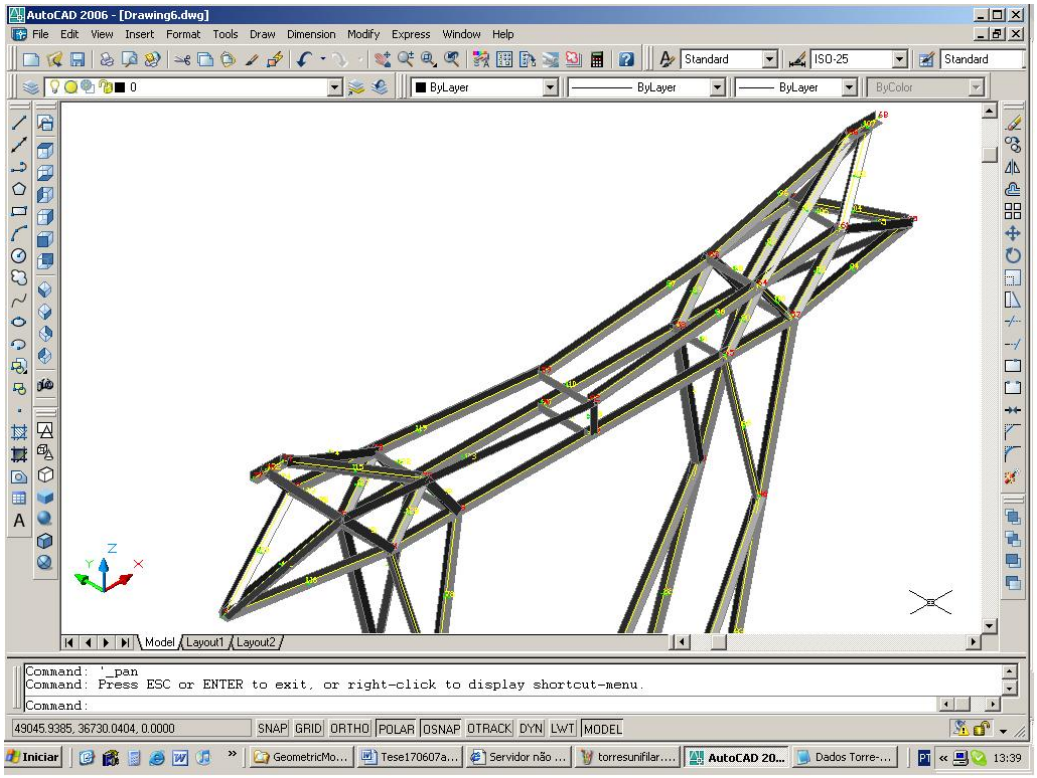

FIGURA 6.15 – Detalhe de torre de transmissão: modelo sólido gerado a partir do *TowerSystem*.

6.4 Aplicações PREMOLD e STEELMOLD

A partir do sistema CAD, *Geometric Modeler,* foram desenvolvidos dois sistemas para a automação do lançamento de estruturas pré-fabricadas: o PREMOLD, para estruturas de concreto pré-moldado, e o STEELMOLD, para estruturas metálicas. Estes sistemas utilizam a funcionalidade do componente *Geometric Modeler*, novas funcionalidades pela implementação de *hot spots* do *framework* REMFrame e o *framework* MFC em sua estruturação. Estes dois sistemas foram desenvolvidos paralelamente dentro de um mesmo componente ARX denominado PREMOLDED (FIGURA 6.16).

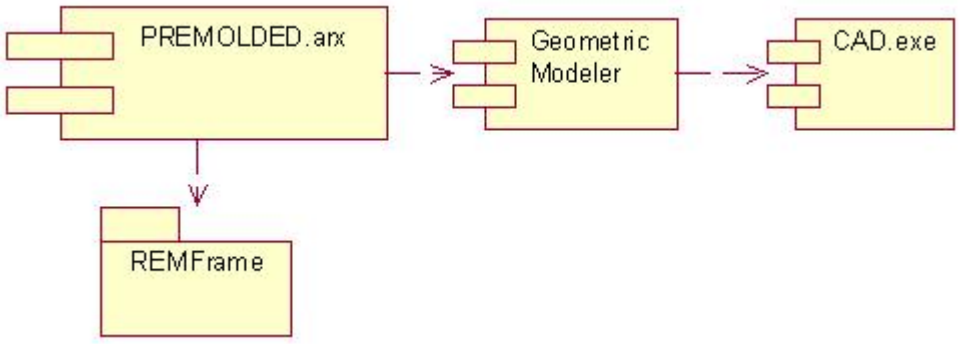

FIGURA 6.16 – Componentes das aplicações PREMOLD e STEELMOLD

Estes sistemas permitem o lançamento automatizado de peças lineares, pilares e vigas, individualmente ou em conjunto, de maneira semelhante à funcionalidade do PREMOLD original (LEITE, 2002). O lançamento dos elementos no modelo é feito com a definição de seções transversais e de materiais. É possível carregar uma biblioteca de seções em tempo de execução, gerar novas seções e inseri-las na biblioteca existente, alterar a seção aplicada a um elemento, etc. Os elementos podem ser inseridos em qualquer direção e segundo pontos de inserção pré-definidos, alinhando os elementos pela esquerda, pela direita, pelo centro geométrico, pela face superior ou pela face inferior. O modelo gerado continua a permitir os mesmos comandos desenvolvidos para o modelador *Geometric Modeler*, ou seja, edições sobre o modelo geométrico, gerar um modelo discretizado a partir do modelo geométrico, inserir carregamentos, restrições nodais, deslocamentos prescritos, liberações nos elementos, etc.

Para ambos os sistemas foram implementadas duas classes para as caixas de diálogo de criação dos elementos estruturais e de uma malha geométrica auxiliar para o lançamento de projetos estruturais modulares, as classes *ElementDialog* e *GeomMeshDialog*, respectivamente. A classe *GeomMeshBuider* implementa a criação da malha geométrica auxiliar. Estas classes foram utilizadas para o desenvolvimento dos dois sistemas. As classes do componente PREMOLDED e seus relacionamentos são mostrados na FIGURA 6.17.

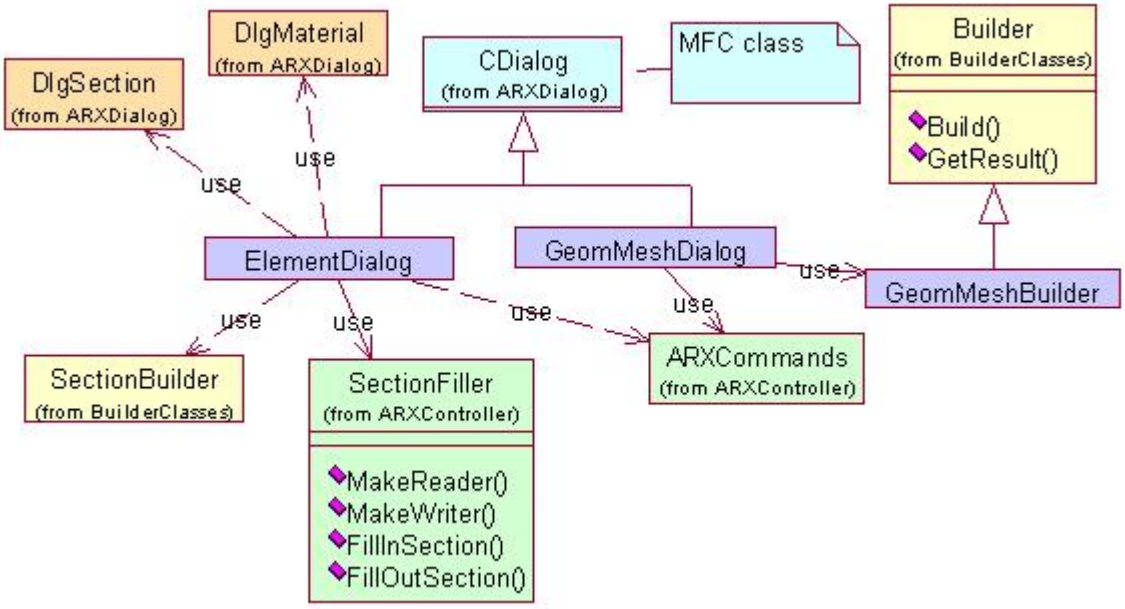

FIGURA 6.17 – Classes do componente PREMOLDED

Para estruturas pré-moldadas em concreto armado as seções dos elementos variam conforme sua função estrutural: viga ou pilar. Para estas estruturas as seções dos pilares são geralmente retangulares, enquanto as seções das vigas possuem formas variadas. Optou-se, então, por separar a modelagem destes elementos em dois comandos, permitindo a variação dos algoritmos de leitura das bibliotecas de seções. Para cada um destes comandos foi criada uma caixa de diálogo como especialização da caixa genérica definida pela classe *ElementDialog*. As seções utilizadas para o sistema PREMOLD foram obtidas do "Manual Técnico: Construções Pré-fabricadas" (PREMO, 1999). A partir dos dados técnicos aí obtidos foi gerada uma tabela de perfis para todas as seções de vigas e de pilares. A classe *PremoSectionFiller* implementa os métodos de leitura e gravação de seções na formatação definida para a tabela de perfis gerada, repassando os dados lidos para a classe *PremoSectionBuilder,* que constrói as poligonais que formam as seções transversais utilizadas pelo sistema PREMOLD. A FIGURA 6.18 mostra a estrutura de classes do sistema PREMOLD.

Para o sistema STEELMOLD foi criado um único comando de lançamento de elementos estruturais, com um algoritmo de inserção de elementos válido tanto para vigas quanto para pilares. Como biblioteca de seções foi utilizada a Lista de Perfis

Soldados USIMINAS (2007), *on line*[25] em sua formatação original. A classe *SteelSectionFiller* implementa os métodos de leitura de dados para a formatação da lista utilizada. A construção da poligonal dos perfis definidos nesta lista é implementada pela classe *IBuilder*.

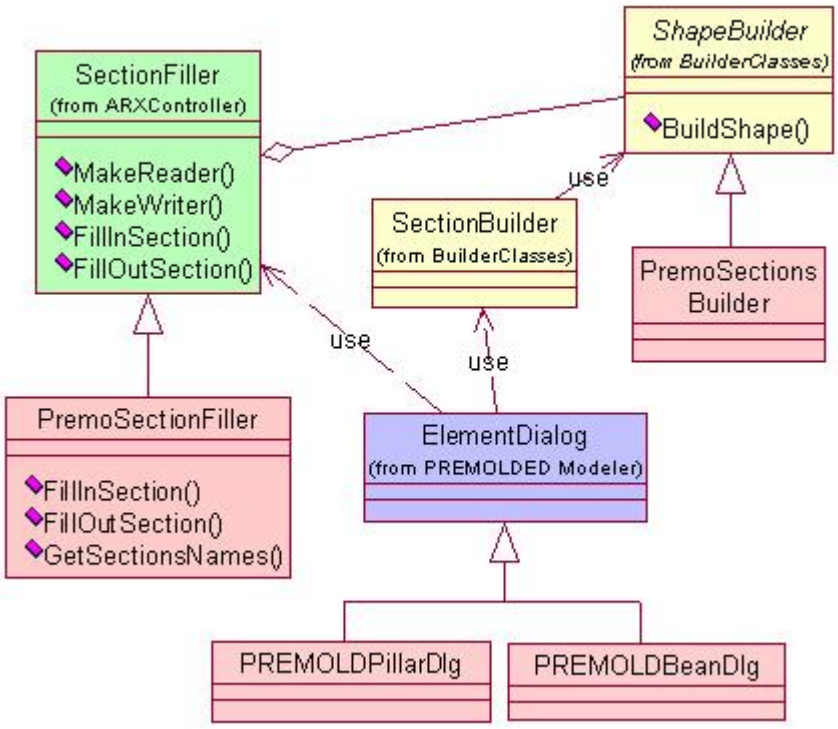

FIGURA 6.18 – Classes da aplicação PREMOLD

[25] http://www.umsa.com.br/ext/perfis_metalicos/soldados/xls/tabela%20soldado.xls

Para a construção das poligonais foi definida uma base comum de dados formada por listas das dimensões parciais verticais e horizontais que compõem as seções. Estas listas devem ser preenchidas pelas classes de leitura de dados e repassadas para as classes *builder*. Esta base comum de dados permite a definição de outras classes builder para diversos tipos de seções transversais, uma vez que apenas a formatação de leitura de dados torna-se dependente da tabela utilizada. Foram então já implementadas classes de construção de poligonais para outras seções metálicas, prevendo-se a utilização de tabelas para perfis dobrados a frio em futuras implementações. As classes definidas para o sistema STEELMOLD são apresentadas na FIGURA 6.19.

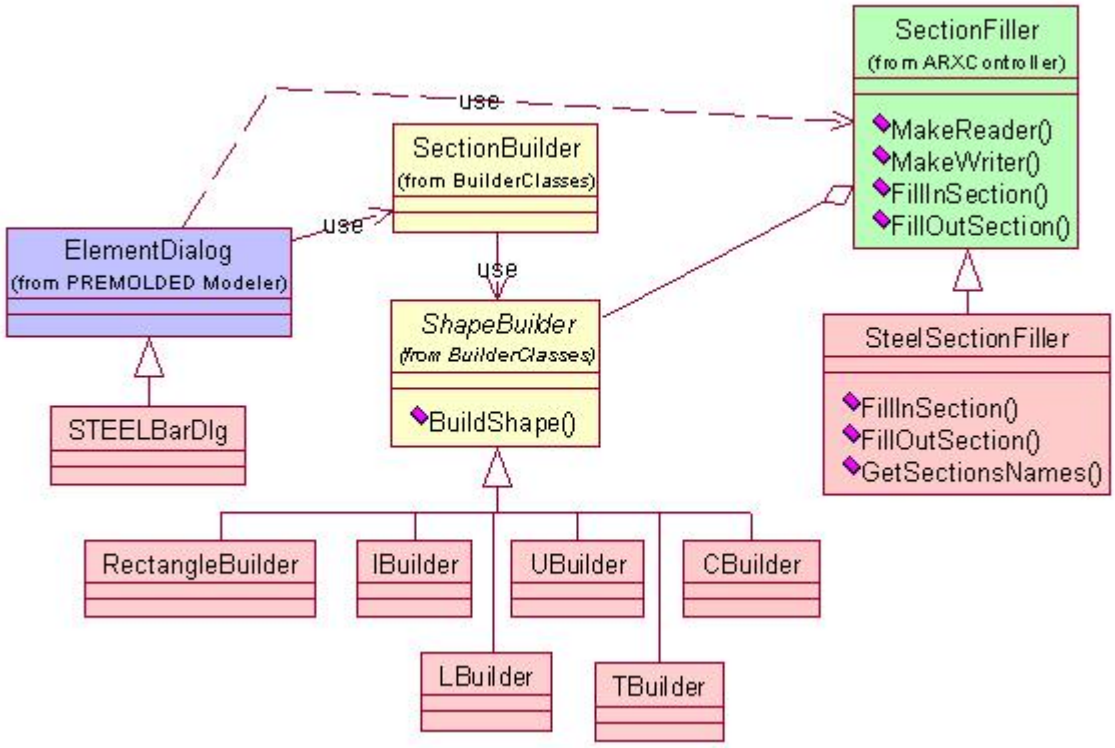

FIGURA 6.19 – Classes da aplicação STEELMOLD

6.4.1 Estudo de Caso

Para exemplificar os modelos gerados pelos sistemas PREMOLD e STEELMOLD, foram lançadas duas estruturas com a mesma geometria unifilar: 103 barras e 64 nós. A estrutura gerada com o modelador PREMOLD recebeu quatro tipos

de seções transversais: um pilar PR40x40 e vigas VTS25x60-PS15, VL60 e viga uma retangular de 40cm x 60cm, definidas no “Manual Técnico: Construções Pré-fabricadas” (PREMO, 1999). A estrutra gerada pelo STEELMOLD recebeu os perfis:CS600X305 para pilares, CVS450X100 e VS600X124 para vigas. As seções transversais destes perfis são contruídas conforme a a Lista de Perfis Soldados USIMINAS (2007), *on line.*[21]. As FIGURA 6.20 a FIGURA 6.23 apresentam os modelos sólido e unifilar das estruturas geradas.

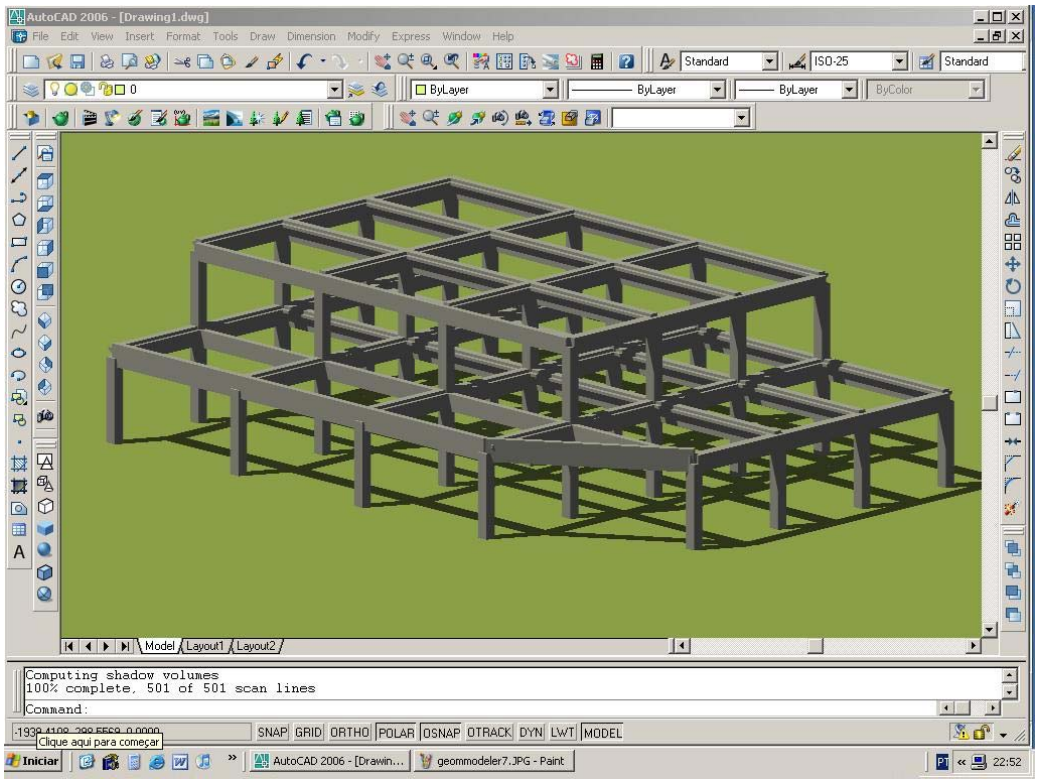

FIGURA 6.20 – Modelo sólido de estrutura em concreto premoldado

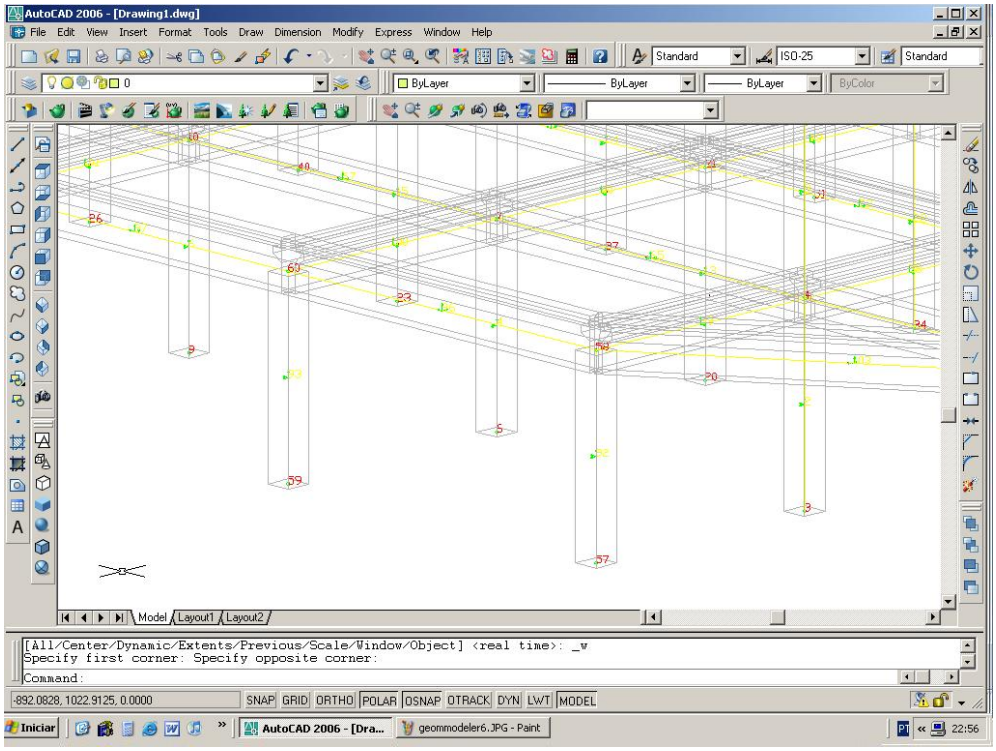

FIGURA 6.21 – Modelo unifilar revestido pelo modelo sólido de estrutura em concreto prémoldado.

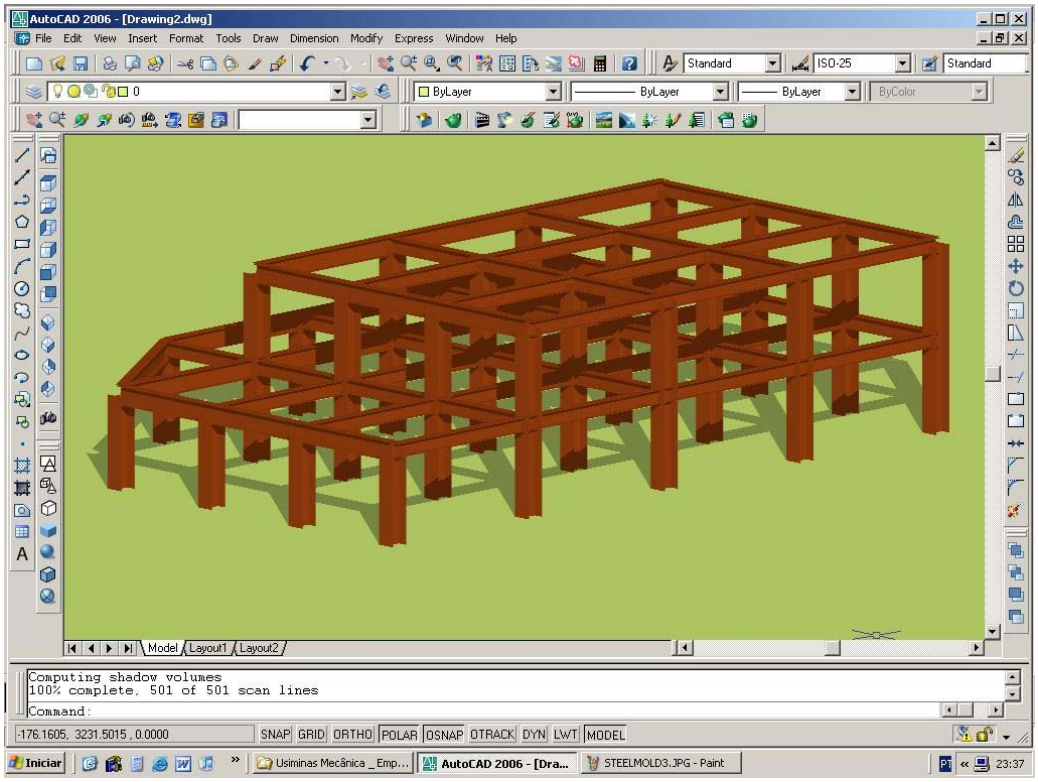

FIGURA 6.22 – Modelo sólido de estrutura metálica

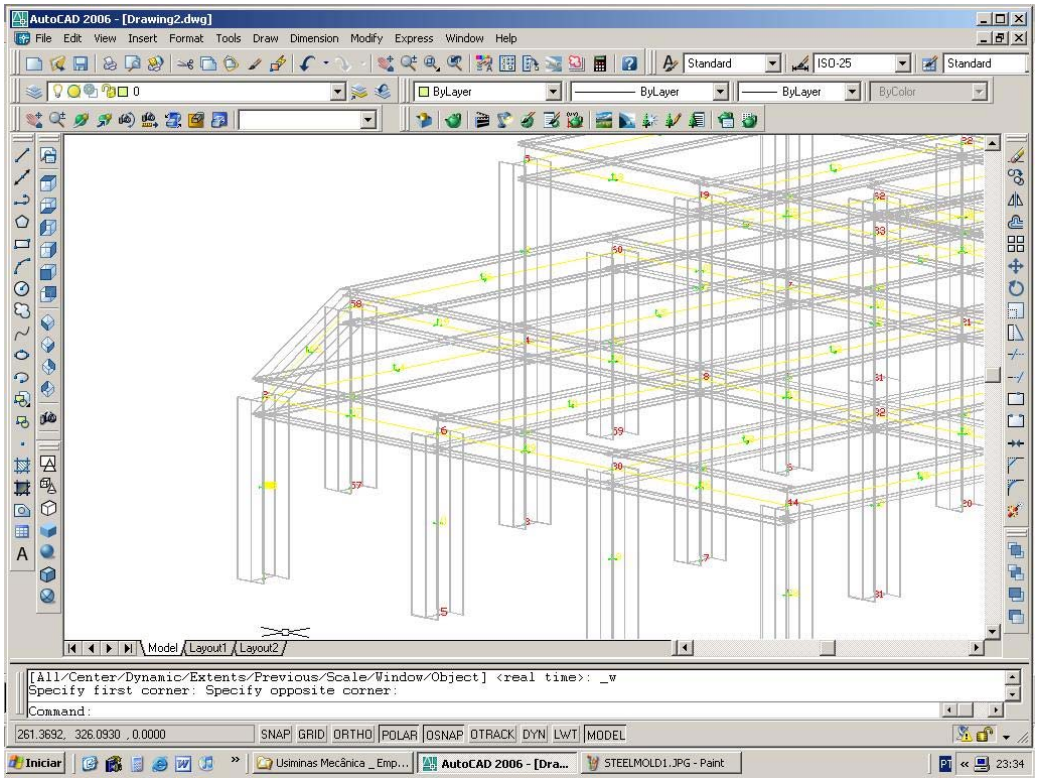

FIGURA 6.23 – Modelo unifilar revestido pelo modelo sólido de estrutura metálica

6.5 Aplicação NLG Analysis

A aplicação *NLG Analysis* foi desenvolvida como um sistema de integração que vinculam o sistema CAD, Geometric *Modeler,* ao sistema de análise NLG (GRECO, 2006).

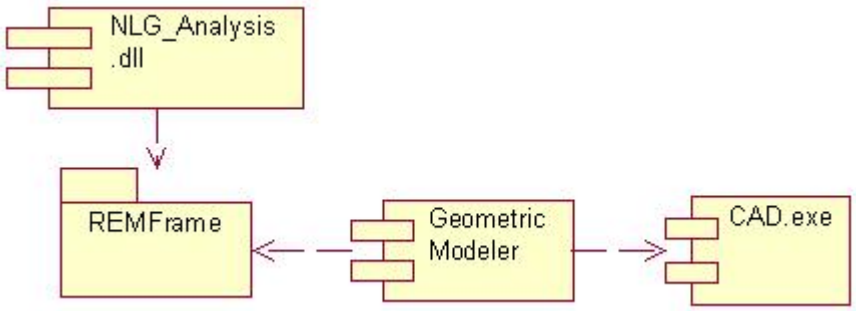

FIGURA 6.24 – Componentes da aplicação *NLG_Analysis*

O sistema é composto pelos componentes *NLG_Analysis, Geometric Modeler,* integrados através do *framework* REMFrame (FIGURA 6.24). No componente *NLG_Analysis* foram definidas as classes de configuração do ambiente (*NLGDomain*) e de gerenciamento de transferência de dados (*NLG_Analysis*). Os objetos destas classes são criados pela classe fábrica *NLGFactory* (FIGURA 6.25).

Para permitir a importação de estruturas previamente definidas e cujos dados estão organizados em arquivos ASCII, cedidos por GRECO para os testes de aplicação, foi implementada a classe *NLG_Analysis.* Esta classe define os métodos para leitura dos dados geométricos e das condições de contorno destas estruturas, gerando um projeto gerenciado pelas classes do *framework*. Este projeto é composto por um modelo geométrico e um modelo de análise. O modelo carregado pelo sistema *NLG_Analysis* pode ser editado pelos comandos do modelador *Geometric Modeler,* podendo ser gravado no formato padrão dos arquivos de persistência definido pelo *framework*. Para exportar um modelo de análise gerenciado pela aplicação *NGL Analysis* para o sistema externo de análise, é necessário definir alguns dados específicos para a formulação utilizada pelo sistema NLG. Por tratar-se de um sistema para análise dinâmica, este deve ser alimentado com o número de passos de carga, a tolerância admitida e a freqüência de impressão de resultados. O sistema informa para um determinado nó da estrutura, sua posição a cada passo de carga e para isso é necessário informar o nó a ser avaliado. Outros dados são: o módulo plástico (K), o módulo de encruamento cinemático (H) e a tensão de escoamento para cada barra analisada. Estes dados são então tratados pela classe *NLGDomain*. Esta classe possui uma configuração padrão que pode ser alterada pelo usuário através do método *Update*.

O caso de uso *Export* Data é apresentado no item 5.3.8. Nesta aplicação as classes *NLGAnalysis,NLGDomain* e *NLGFactory* fazem a implementação dos *hot spots* definidos pelas classes *DataManager, Domain* e *AnalysisDataFactory,* respectivamente. A classe de controle recebe como atributo um objeto da classe *NLGFactory*. Este objeto é o responsável por instanciar os objetos de formatação de transferência de dados e de domínio através das classes *NLG_Analysis* e *NLGDomain*. O método *Update,* implementado na classe de domínio *NLGDomain,* utiliza a classe *NLGDialog* para obter a configuração necessária para o sistema. A classe

NLG_Analysis implementa a formatação para exportação dos dados e ainda instancia, de forma correta, os documentos de entrada e de saída de dados.

Após a análise do modelo estrutural, o modelador *Geometric Modeler* é utilizado para representar a deformada da estrutura e os diagramas de esforços nas barras. O sistema NLG gera três arquivos de saída: um arquivo contém as posições dos elementos nos passos de carga estabelecidos pela freqüência informada ao sistema, outro arquivo informa as posições de um determinado nó em cada passo de carga e o terceiro arquivo apresenta os esforços nas extremidades das barras, também na freqüência informada. A classe *NLG_Analysis*, seleciona o arquivo de saída conforme o tipo de representação solicitado pelo modelador, executa a leitura de dados e o preenchimento do modelo com os dados obtidos. O modelo, após as alterações realizadas solicita a suas visualizações que se atualizem, obtendo a representação dos dados solicitada pelo usuário. O diagrama de seqüência do caso de uso *Import Data* foi apresentado no item 5.3.8.

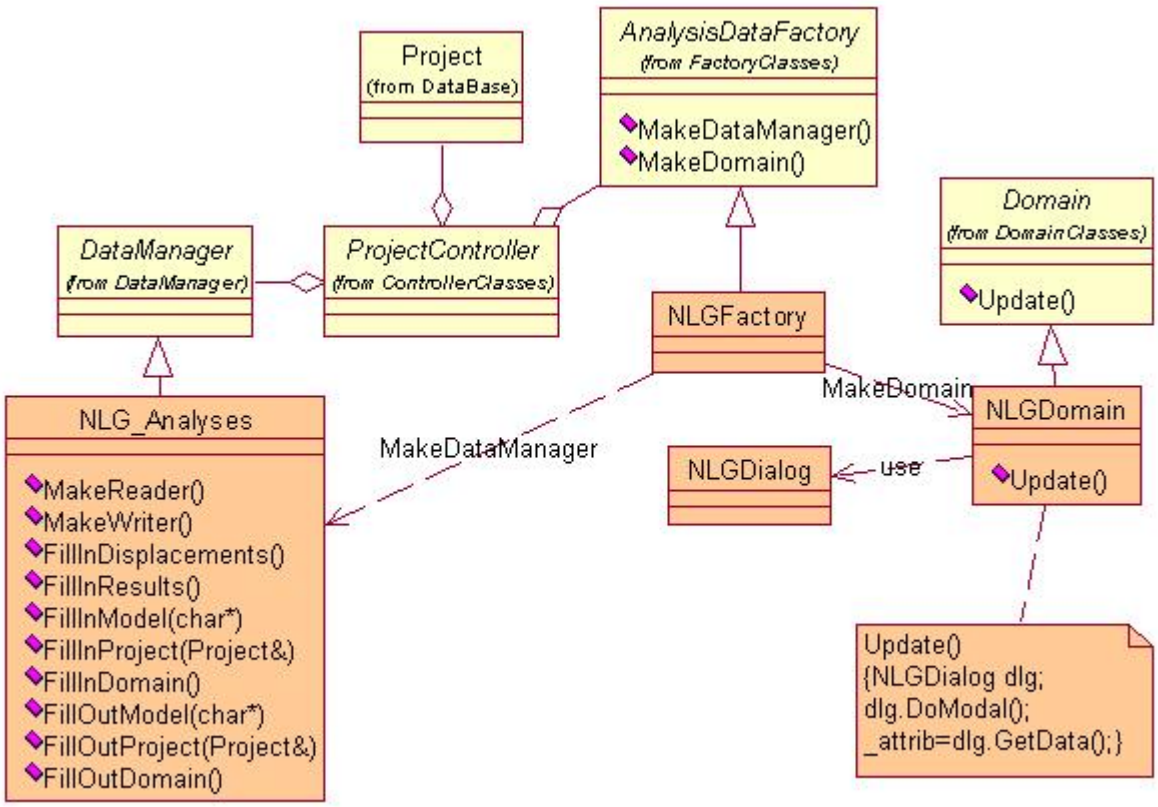

FIGURA 6.25 – Classes do componente *NLG_Analysis*

6.5.1 Estudo de Caso

Para exemplificar o funcionamento da aplicação *NLG_Analysis* foram rodados três exemplos de treliças espaciais: a cúpula *Star* e a cúpula Schewdeler apresentadas por GRECO (2006), além de uma cúpula geodésica proposta por FÜLLER (1975). A entrada de dados de geometria das estruturas e de condições de contorno pode ser realizada tanto pelo caso de uso *Import Data* quanto pelos comandos do modelador *Geometric Modeler*.

A cúpula *Star* possui 24 elementos e 13 nós. Foram aplicados 500 passos de deslocamentos de 0,1cm no nó central da cúpula. A FIGURA 6.26 mostra a estrutura revestida por uma seção transversal e a deformada (em azul) referente ao passo 100. A FIGURA 6.27 mostra os elementos do modelo refinado (em amarelo) e a deformada referente ao passo 100 (em azul), enquanto a FIGURA 6.28 mostra os diagramas de esforços normais (em vermelho) nos elementos referentes ao passo 100.

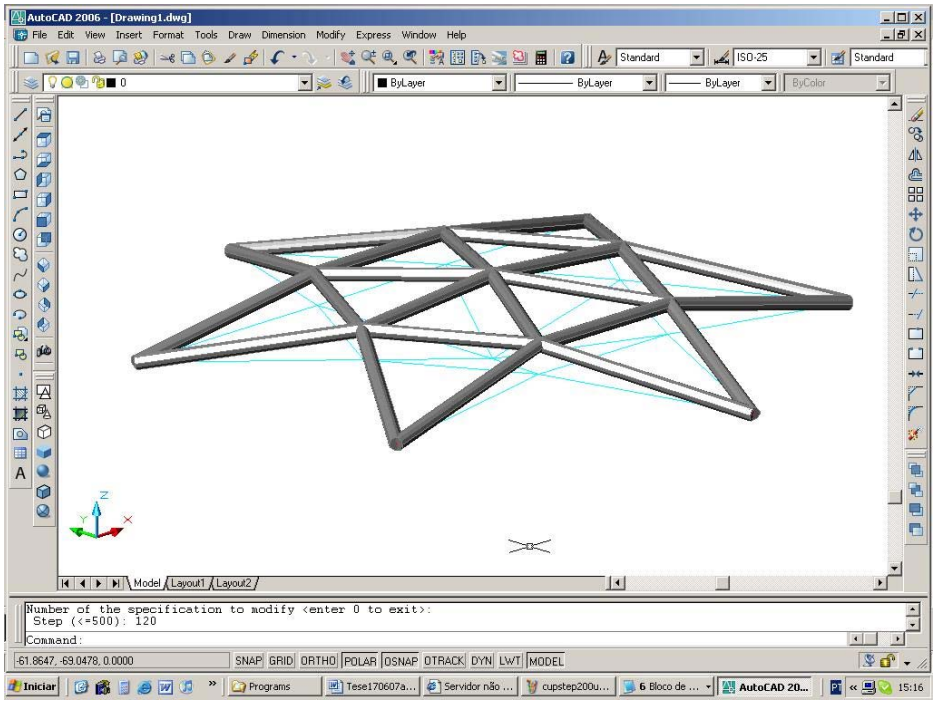

FIGURA 6.26 – Cúpula Star: modelo sólido e deformada do passo de carga 100

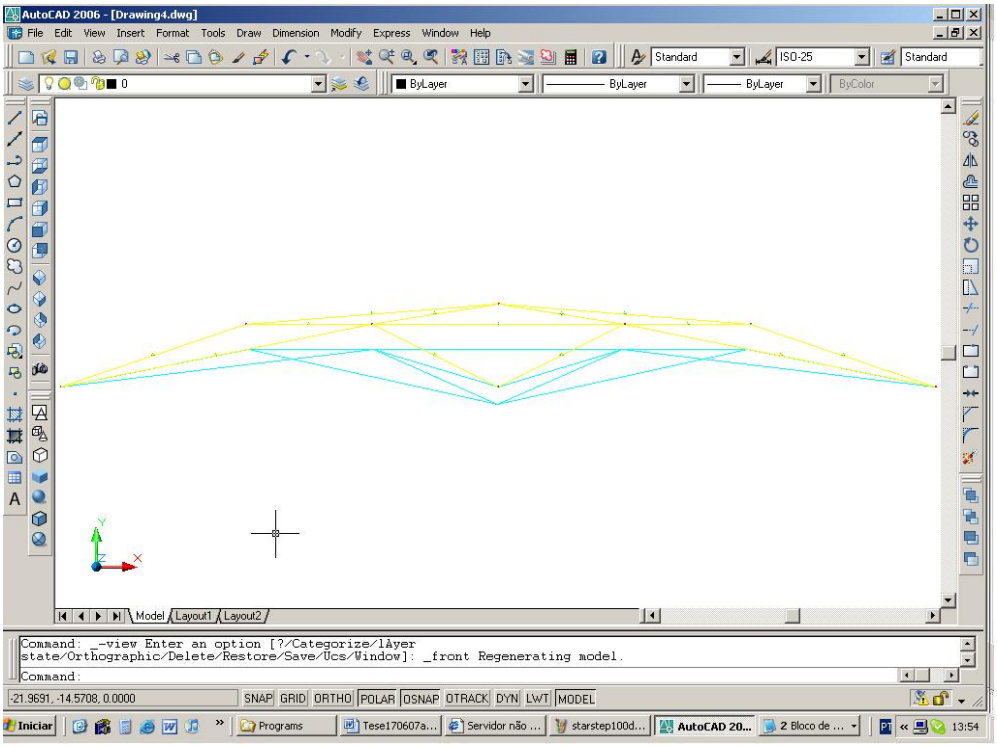

FIGURA 6.27 - – Cúpula Star: modelo unifilar e deformada do passo de carga 100, vista lateral

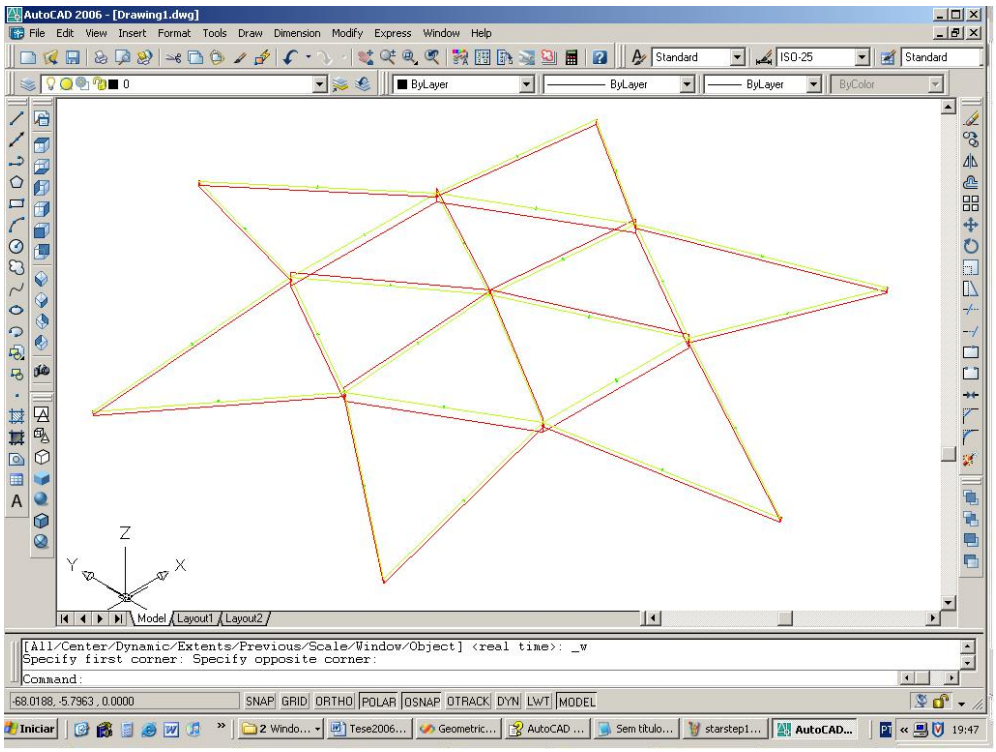

FIGURA 6.28 - – Cúpula Star: modelo unifilar e esforços do passo de carga 100

A cúpula Schewdeler possui 264 elementos e 97 nós. São aplicados 8000 passos de deformações de 0,1cm no nó central, no topo da cúpula, cuja altura é de 4,58 metros. A FIGURA 6.29 mostra o modelo refinado da estrutura com representação das barras em amarelo, com identificação dos nós em vermelho e dos eixos locais de cada barra em verde. Para as estruturas treliçadas os modelos geométrico e refinado são coincidentes. A FIGURA 6.30 à FIGURA 6.33 mostram a deformada da estrutura (em azul) e os diagramas de esforços (em vermelho) no passo 4000 enquanto a FIGURA 6.34 à FIGURA 6.37 apresentam a deformada da estrutura (em azul) e os diagramas de esforços (em vermelho) no passo 8000.

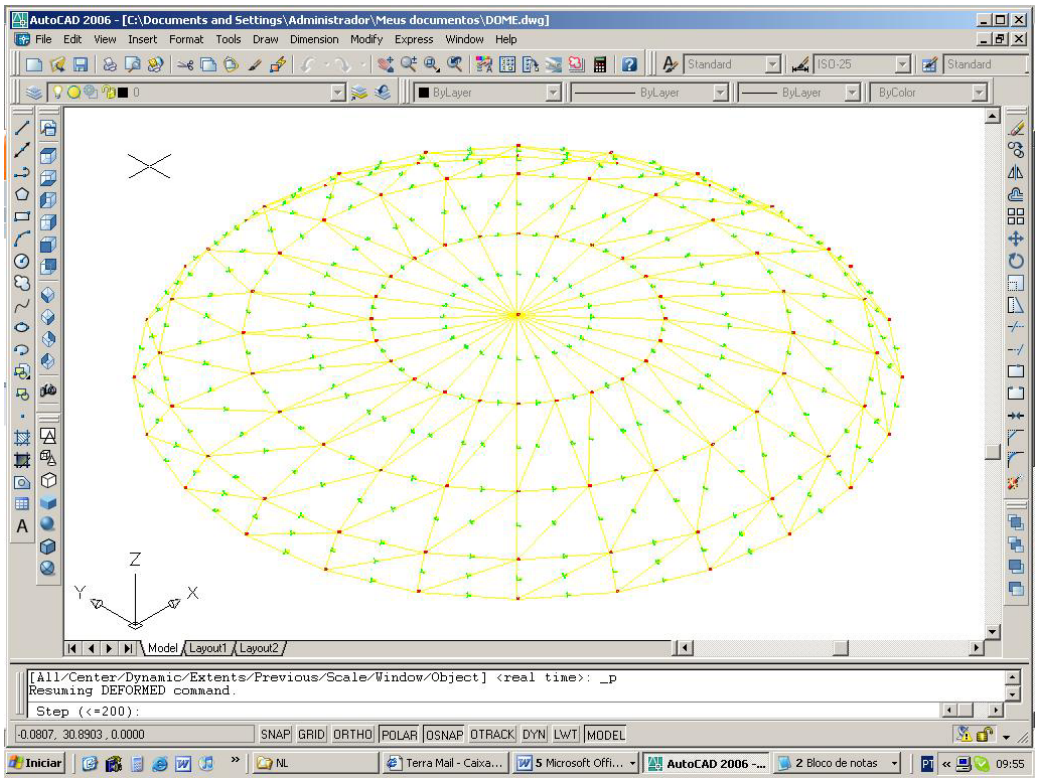

FIGURA 6.29 – Cúpula de Schewdeler: Modelo unifilar.

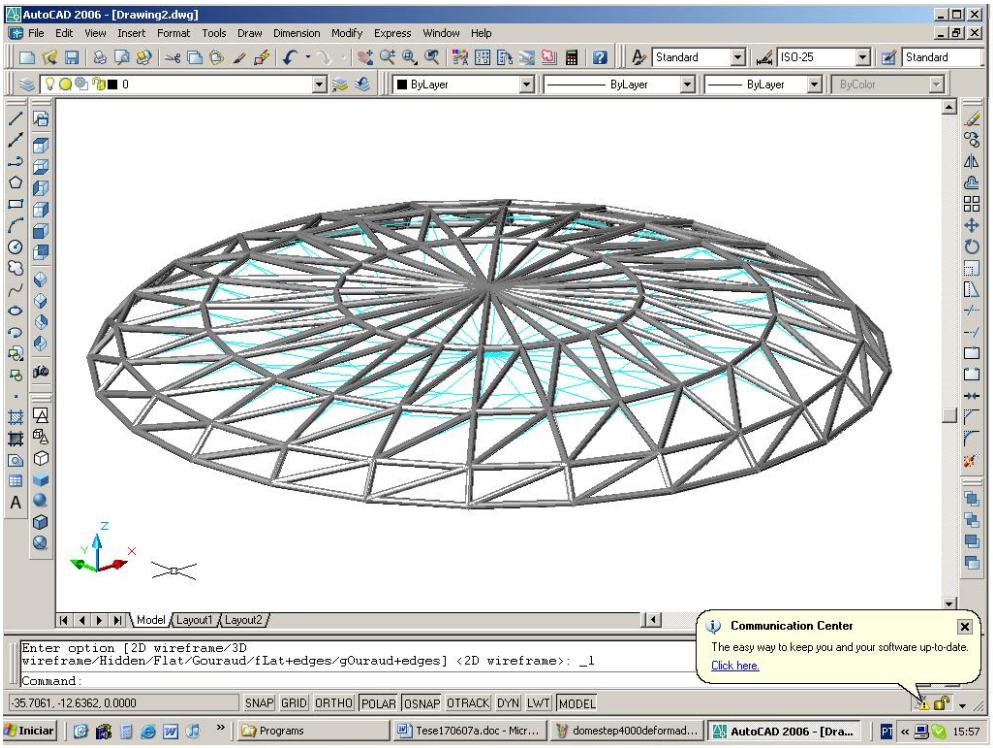

FIGURA 6.30 - Cúpula de Schewdeler: Modelo sólido e deformada no passo 4000.

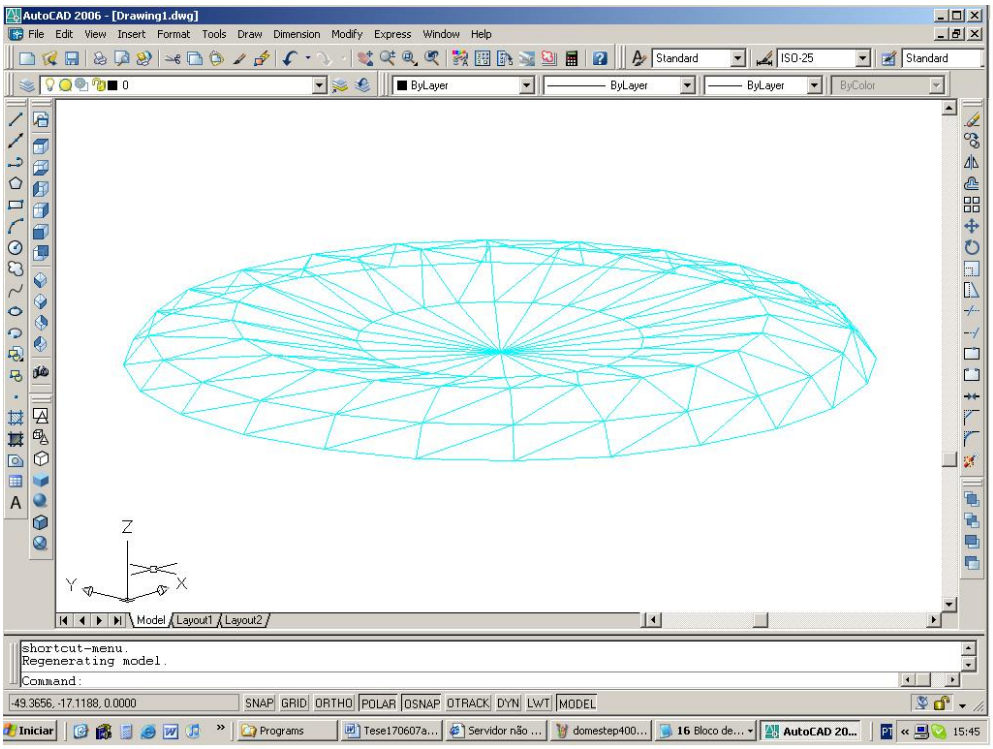

FIGURA 6.31 - Cúpula de Schewdeler: Deformada no passo 4000.

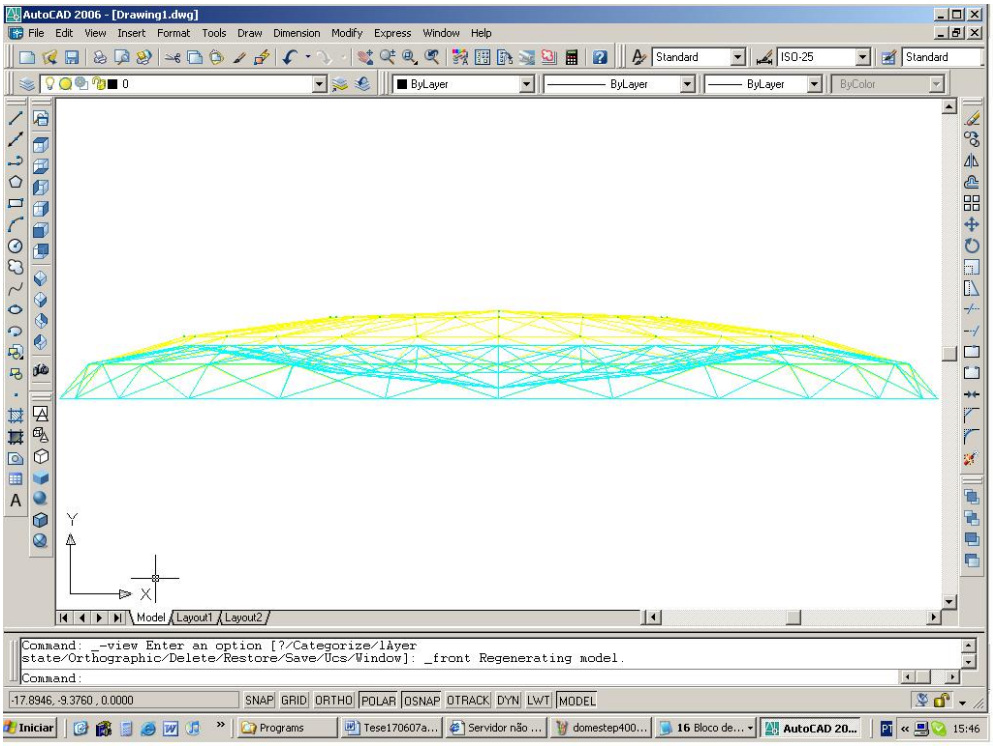

FIGURA 6.32 - Cúpula de Schewdeler: Modelo unifilar e deformada no passo 4000 (vista lateral)

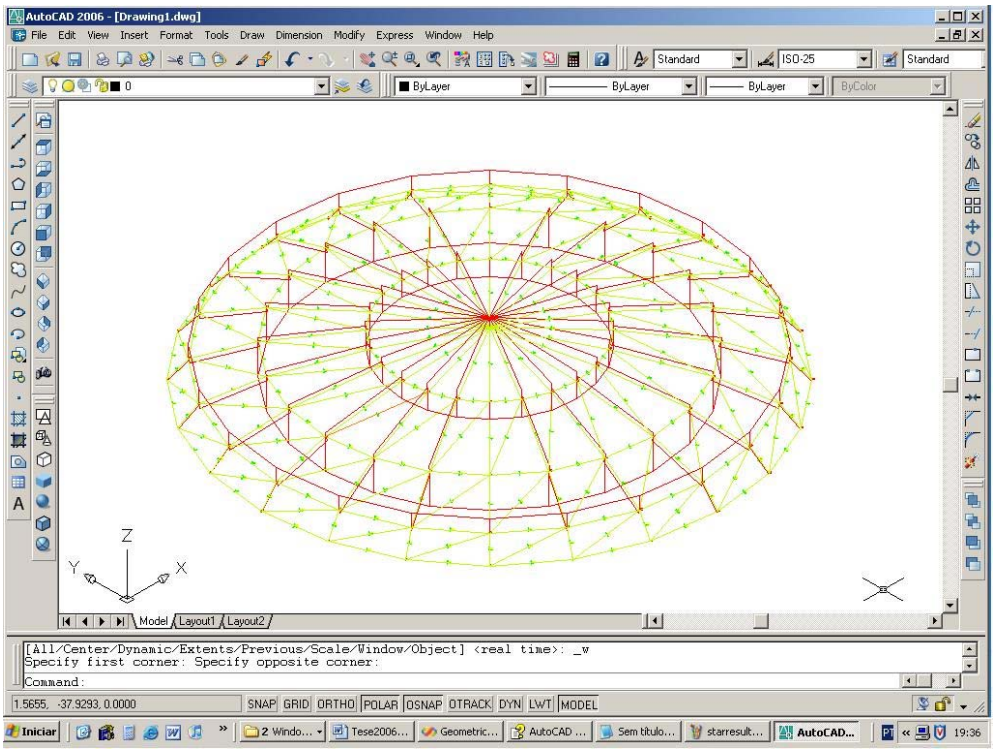

FIGURA 6.33 - Cúpula de Schewdeler: Modelo unifilar.e gráficos de esforços no passo 4000

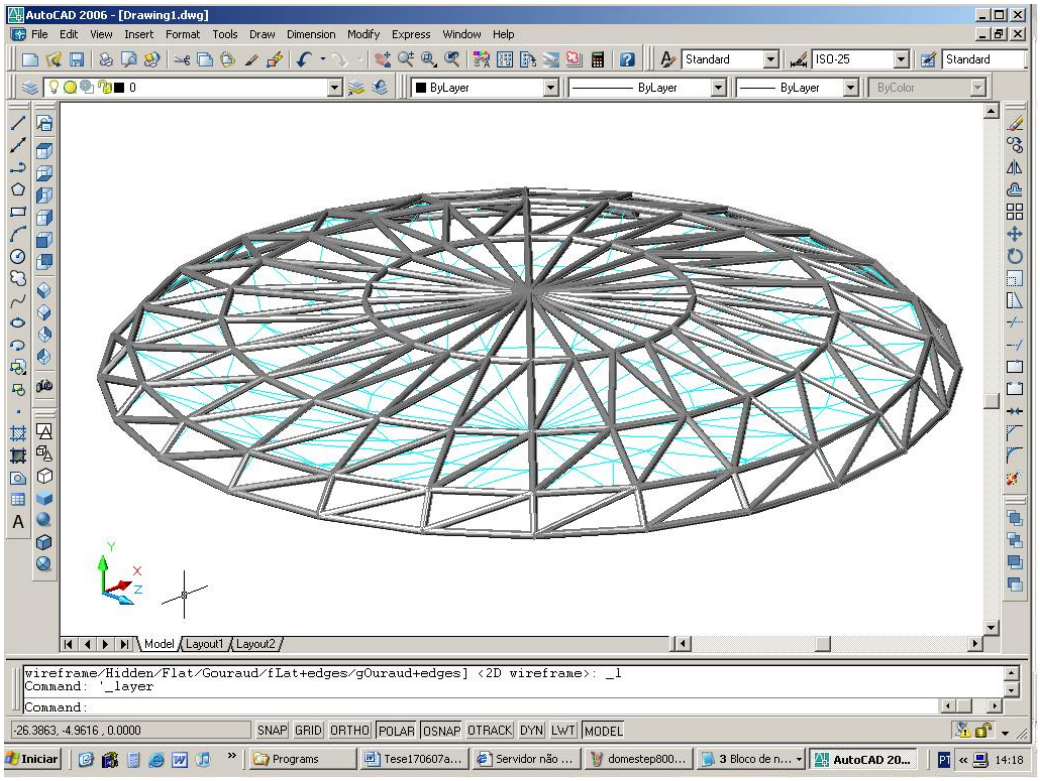

FIGURA 6.34 - Cúpula de Schewdeler: Modelo sólido e deformada no passo 8000.

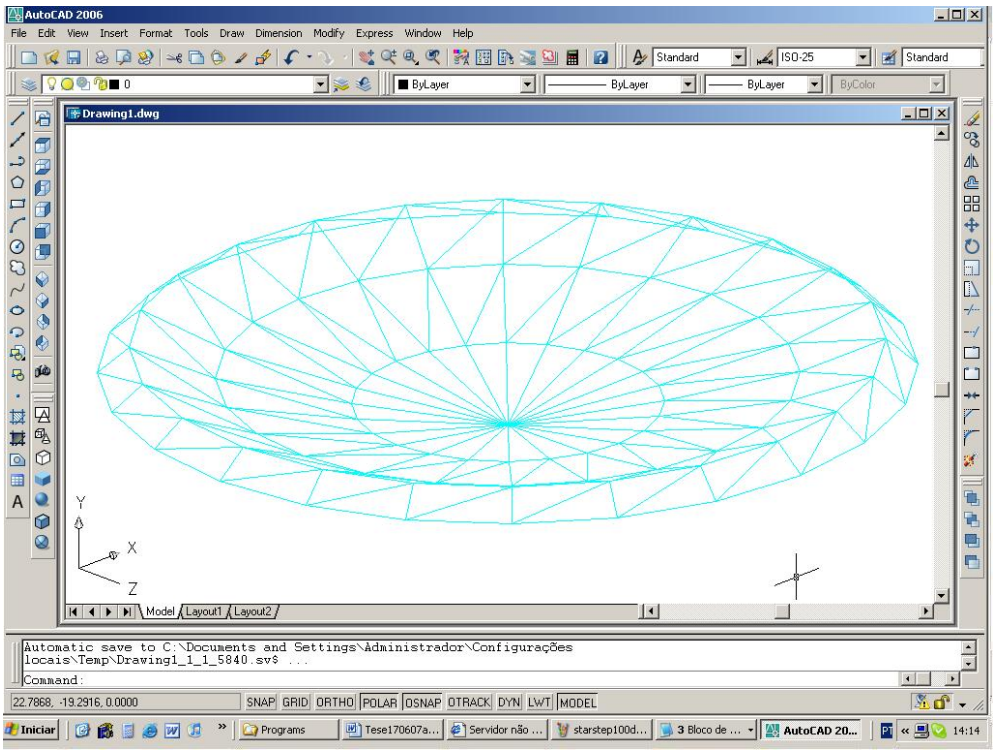

FIGURA 6.35 - Cúpula de Schewdeler: deformada no passo 8000.

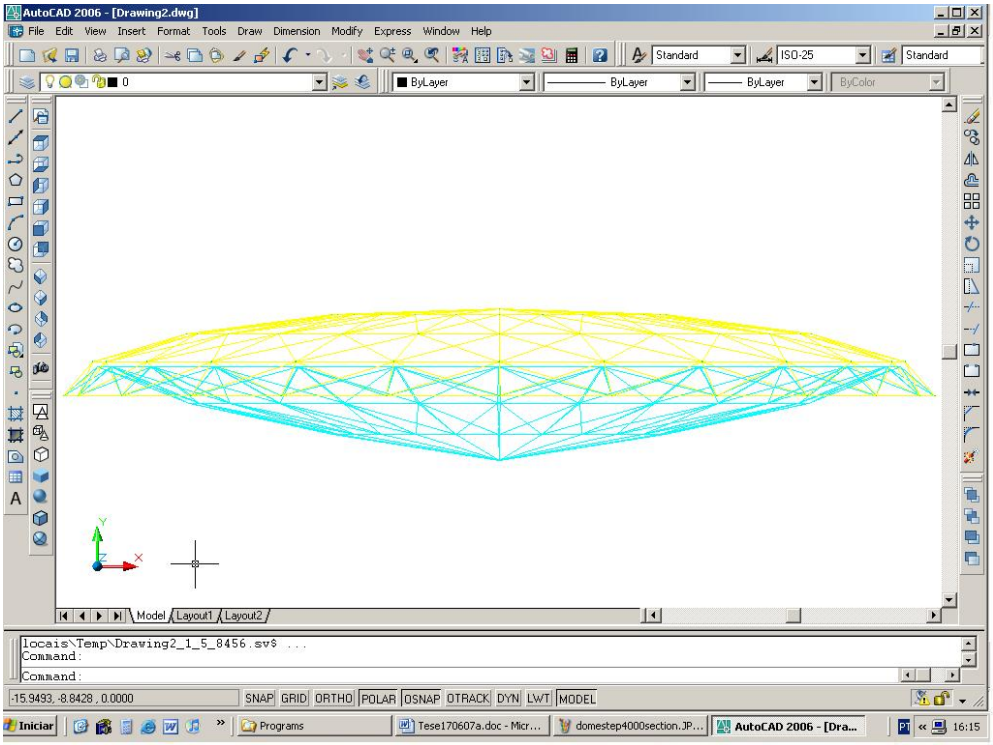

FIGURA 6.36 - Cúpula de Schewdeler: modelo unifilar e deformada no passo 8000 (vista lateral)

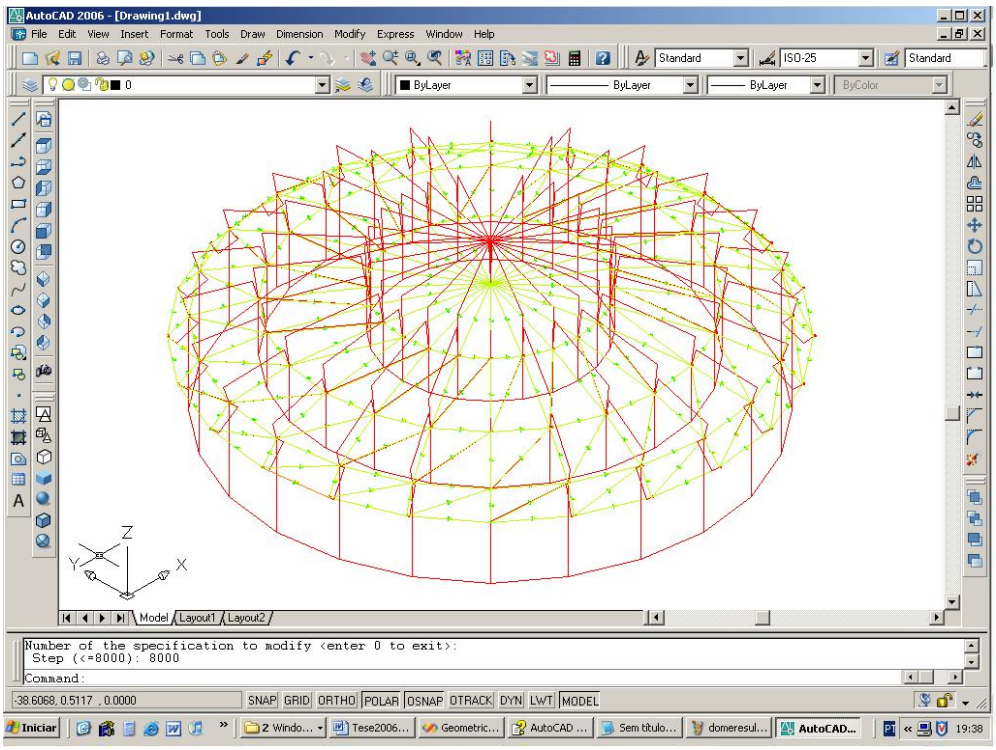

FIGURA 6.37 - Cúpula de Schewdeler: modelo unifilar e diagrama de esforços no passo 8000

A cúpula geodésica de FÜLLER possui 55 elementos e 26 nós. São aplicados 200 passos de carga de -1KN no nó central no topo da cúpula, na direção do eixo Z. A FIGURA 6.38 mostra o modelo sólido da estrutura e a deformada, em azul, para o passo de carga 200. A FIGURA 6.39 e a FIGURA 6.40 apresentam o modelo unifilar, em amarelo, a deformada, em azul, e os diagramas de esforços, em vermelho, no passo 100 enquanto a FIGURA 6.41 e a FIGURA 6.42 apresentam o modelo unifilar, a deformada e os diagramas de esforços no passo 200.

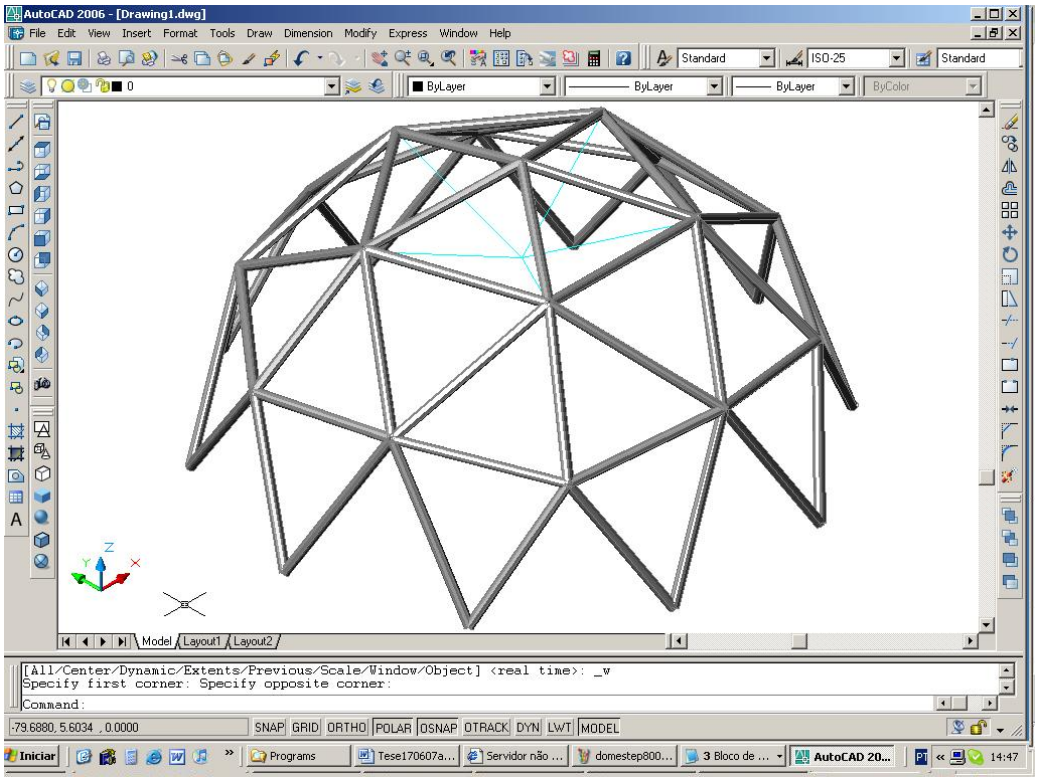

FIGURA 6.38 – Cúpula geodésica: Modelo sólido e deformada no passo 200

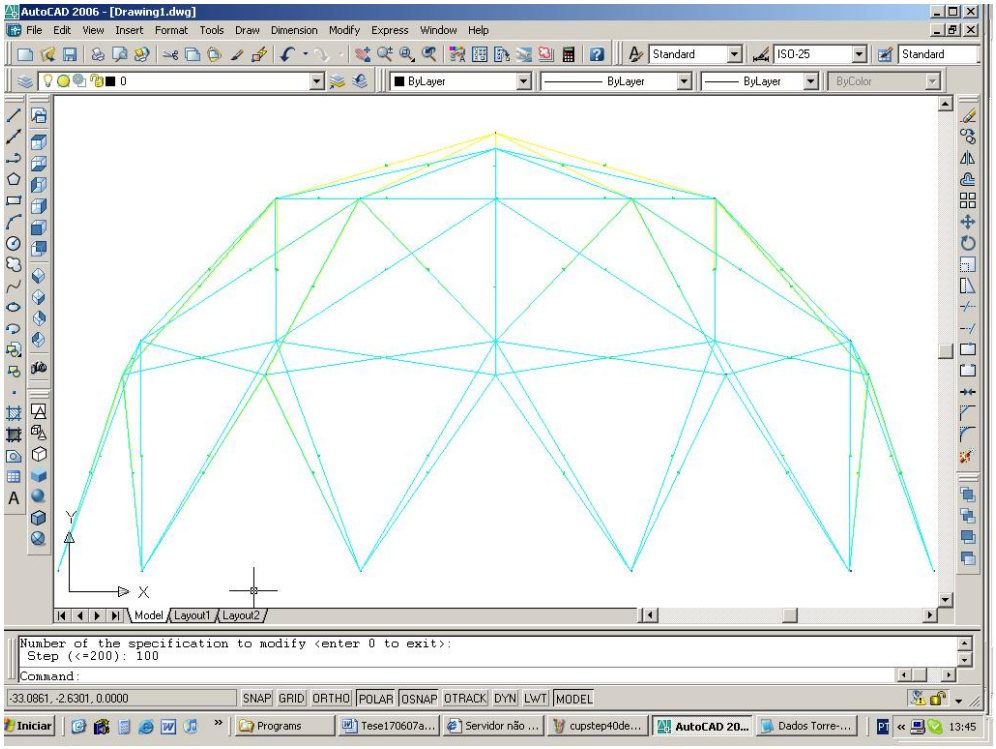

FIGURA 6.39 - Cúpula geodésica: Modelo unifilar e deformada no passo 100

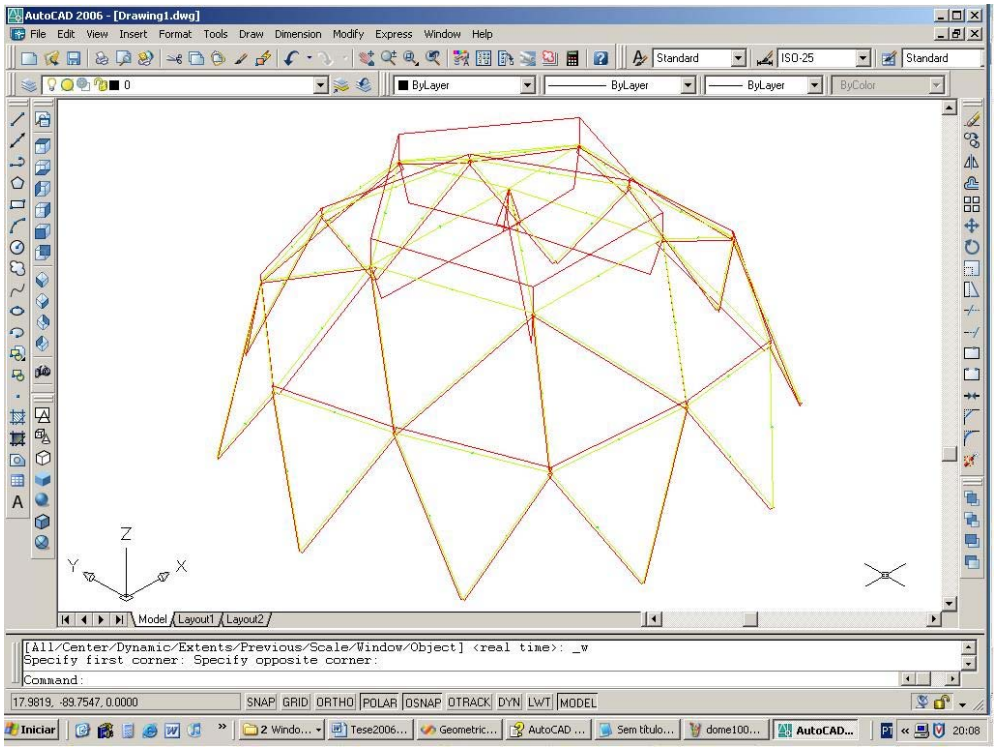

FIGURA 6.40 Cúpula geodésica: Modelo unifilar e diagrama de esforços no passo 100

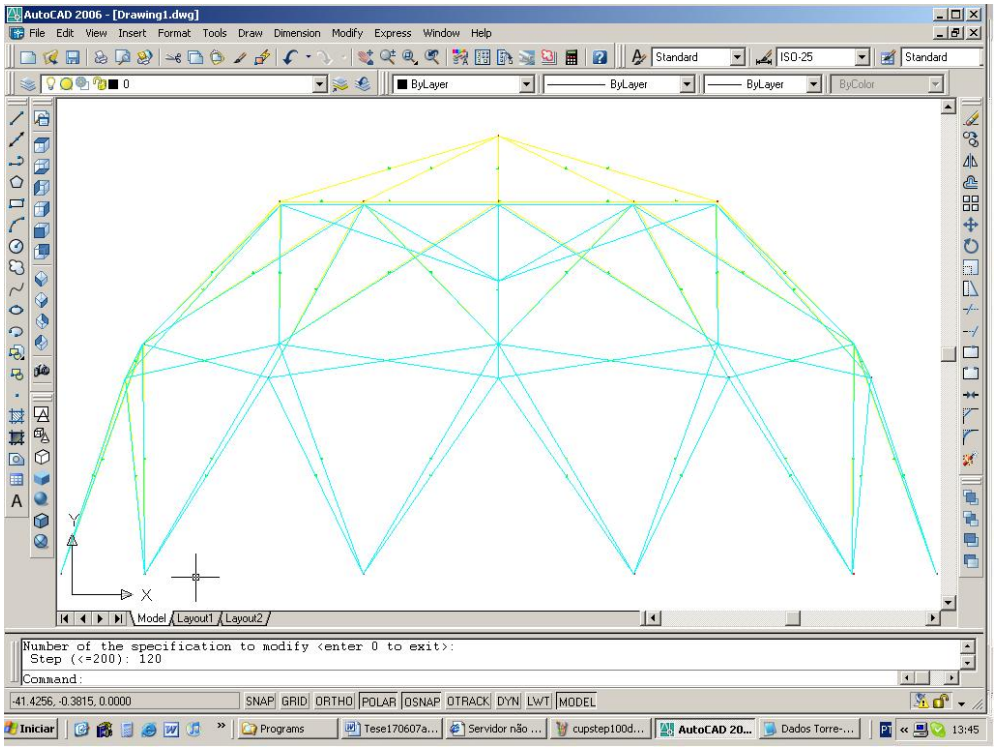

FIGURA 6.41 - Cúpula geodésica: Modelo unifilar e deformada no passo 200

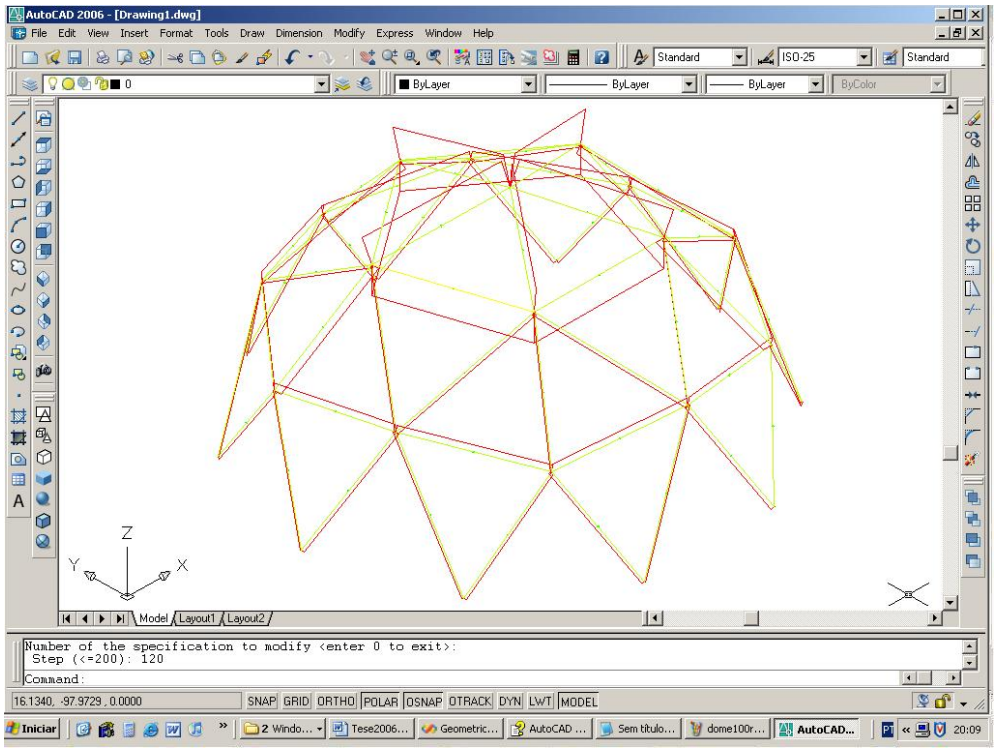

FIGURA 6.42 - Cúpula geodésica: Modelo unifilar e diagrama de esforços no passo 200

6.6 Aplicação PLANLEP Analysis

A aplicação *PLANLEP Analysis* foi desenvolvida, de modo semelhante à aplicação *NLG Analysis* como um sistema de integração. Nesta aplicação são integrados o sistema STEELMOLD e o sistema de análise PLANLEP (LAVALL[18] *apud* SILVA R., 2004).

A organização do sistema, sua arquitetura e seu funcionamento são bastante semelhantes ao sistema *NLG Analysis*, uma vez que são duas aplicações dentro de um mesmo domínio (FIGURA 6.43 e FIGURA 6.44). Para esta aplicação foram definidas as classes *PLANLEP_Analyse, PLANLEPDomain* e*, PLANLEPFactory* que implementam os *hot spots DataManager, Domain e AnalysisDataFactory,* respectivamente. A classe *PLANLEPFactory* instancia os objetos das classes *PLANLEP_Analyse* e *PLANLEPDomain,* e o método *Update* da classe *PLANLEPDomain* utiliza a classe *PLANLEPDialog* para obter os dados de configuração do sistema que não podem ser obtidos diretamente dos dados do modelo. Para o sistema PLANLEP, o dado material engloba dados abstraídos pelas classes *Material* e *Section* do *framework*. Assim, as características físicas do objeto do tipo *Material* e os dados geométricos da seção transversal, representados por objetos do tipo *Section* são obtidos, tratados e formatados pela classe *PLANLEP_Analysis* para gerar os dados de materiais necessários à analise pelo sistema PLANLEP.

Os dados de configuração de domínio para o sistema PLANLEP, tratados pela classe *PLANLEPDomain,* e informados pelo usuário através da classe *PLANLEPDialog,* são: o número de incrementos de carga, o número de fatias da seção transversal, o número de condições de contorno, o algoritmo de saída empregado, o número de graus de liberdade por nó. Para cada incremento de carga é permitido individualizar o número de iterações, o tipo de saída, o percentual de carga permanente, o percentual de carga variável e o percentual de carga de vento e a tolerância a serem aplicados. É possível ainda alterar estes dados para um intervalo de incrementos. A classe *PLANLEPDialog* disponibiliza ainda dois campos para a configuração de fatores de escalas que são

utilizados pela classe PLANLEP_*Analysis* para tratar os dados de deslocamentos e de esforços obtidos da análise antes de inseri-los no modelo, permitindo a adequação dos diagramas gerados à ordem de grandeza dos resultados obtidos na análise.

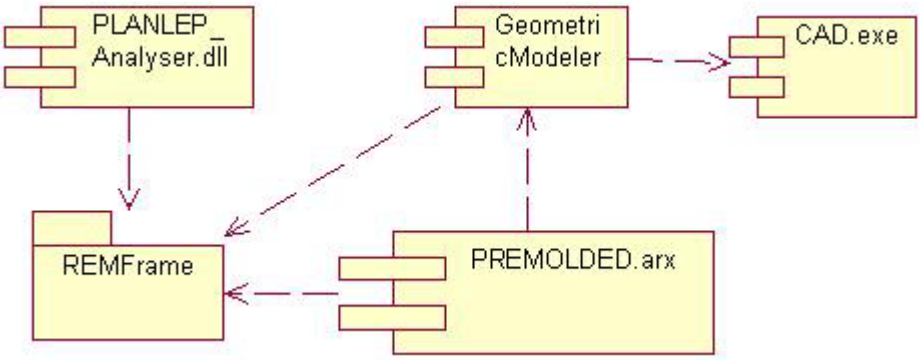

FIGURA 6.43 - Componentes da aplicação *PLANLEP_Analysis*

O sistema PLANLEP define um material pelo conjunto de propriedades físicas e de uma seção transversal dividida em N fatias. Pelos dados geométricos da seção definida através do sistema STEELMOLD, o método *FillOutModel* da classe *PLANLEP_Analysis* obtém os dados de propriedades físicas dos materiais utilizados e conjuga estes com a seção transversal do elemento. Uma rotina implementada por esta classe automatiza a distribuição e o ajustamento das N fatias da seção transversal entre a mesa inferior, a alma e a mesa superior, informando as dimensões de base e altura para cada fatia.

Como o sistema PLANLEP permite a aplicação de fatores de ponderação de cargas, classificando estas em cargas permanentes, variáveis e de vento, a implementação desta aplicação permitiu testar e aprimorar o tratamento dados aos casos de carregamento e a utilização da classe *LoadMethod* para definição dos tipos de cargas e seus fatores de ponderação.

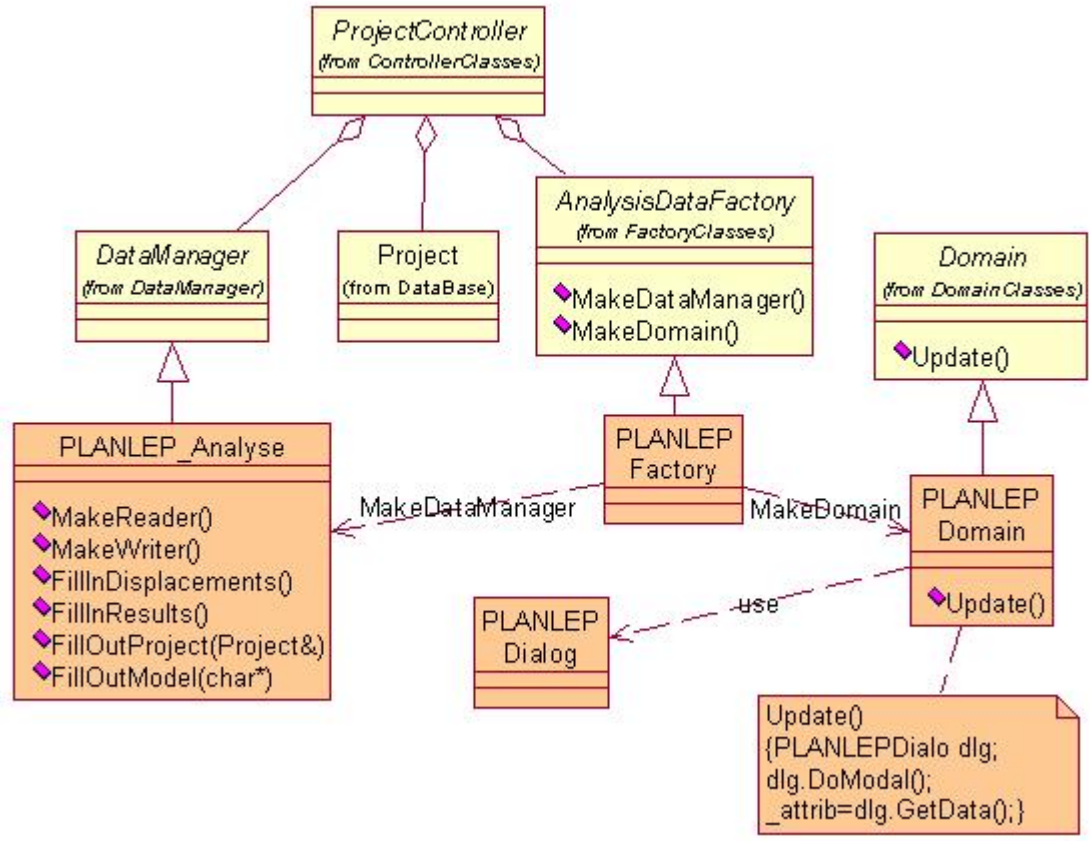

FIGURA 6.44 - Classes do componente *PLANLEP_Analysis*

Os fluxos de dados implementados seguem o mesmo esquema geral adotado para a aplicação *NLG Analysis*, ou seja, as diferenças de implementação para as duas aplicações se encontram exatamente na definição dos *hot spots,* tendo sido aproveitado o núcleo do *framework* sem grandes alterações. Foram apenas realizadas melhorias nas definições dos casos de carregamento juntamente com a aplicação de métodos de carregamento, testado pela primeira vez por esta aplicação.

O sistema PLANLEP apresenta como dados de saída os deslocamentos dos nós, as reações nodais, e os esforços nas extremidades dos elementos, além das deformações plásticas da seção transversal para cada incremento. O sistema *PLANLEP_Analysis* faz a leitura das deformações da estrutura por incremento, buscando a iteração que apresenta convergência de dados, apresentando o diagrama da estrutura deformada. Para

cada iteração também são lidos os dados dos esforços nas extremidades dos elementos, sendo gerados os diagramas de esforços correspondentes.

6.6.1 Estudo de Caso

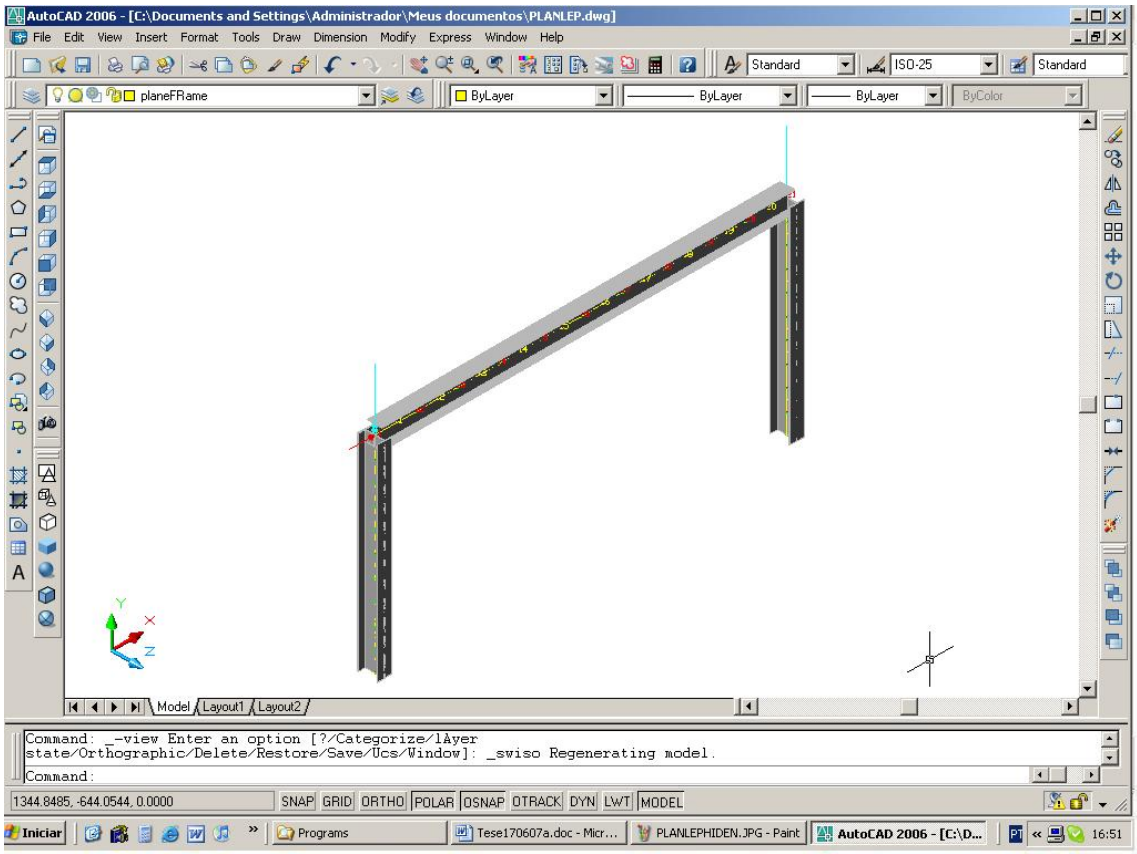

FIGURA 6.45 – Modelo sólido de pórtico de um pavimento e um vão com cargas aplicadas.

O sistema PLANLEP_*Analysis* foi testado através do exemplo de um pórtico não contraventado de um andar e um vão apresentado por SILVA e LAVALL (2005). A FIGURA 6.45 mostra um pórtico de nós rígidos. Os autores assim definem o problema: "Os pilares e a viga são constituídos pelos perfis soldados CS 400x137 e CVS 500x217, respectivamente. O pórtico apresenta uma altura de 5 metros e vão de 10 metros e está sujeito a duas forças verticais P e a uma força horizontal αP no seu topo, sendo α=0,10. O aço apresenta uma tensão de escoamento σy= 35kN/cm2 e módulo de elasticidade E= 20500 kN/cm2**.** Para implementação dos dados do programa, as seções dos perfis foram divididas em 20 fatias, sendo uma fatia para cada mesa e 18 para a alma. As barras foram divididas em 10 elementos de igual comprimento e as forças verticais e

horizontal foram aplicadas de forma incremental. As cargas são dadas em função da carga P_y dos pilares, sendo $Py=A_c\sigma_y= 6090kN$.”

A FIGURA 6.8 mostra a definição do material utilizado neste exemplo.

A FIGURA 6.46 mostra a caixa de diálogo para a definição dos dados de domínio do sistema requeridos pela classe *PLANLEPDomain*. A FIGURA 6.47 apresenta a deformada da estrutura, em azul, para o incremento de carga de número 39 e a FIGURA 6.48 à FIGURA 6.50 mostram os esforços nos elementos, em vermelho, para o mesmo incremento de carga.

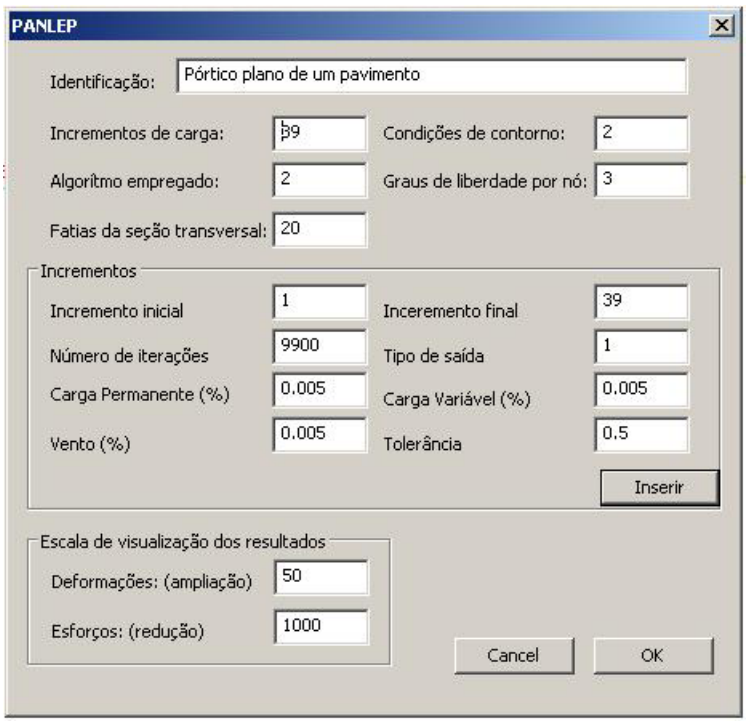

FIGURA 6.46 – Caixa de diálogo para preenchimento de dados de domínio do sistema PLANLEP (dados do exemplo de pórtico de um pavimento e um vão)

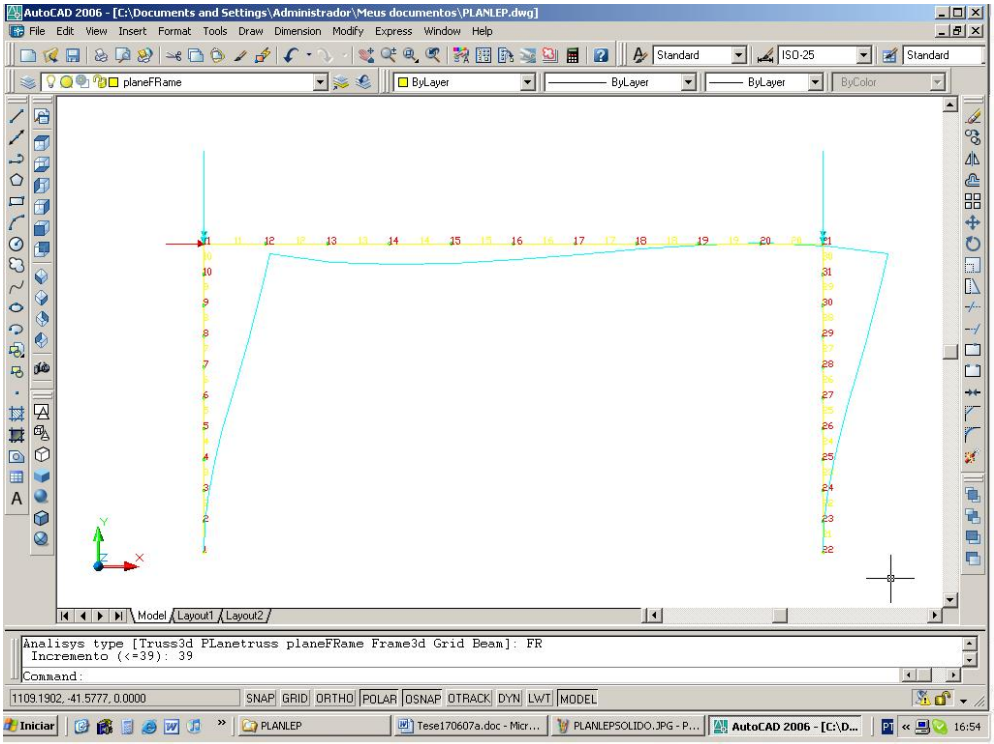

FIGURA 6.47 – Modelo refinado de pórtico de um pavimento e um vão com deformada para o incremento de número 39

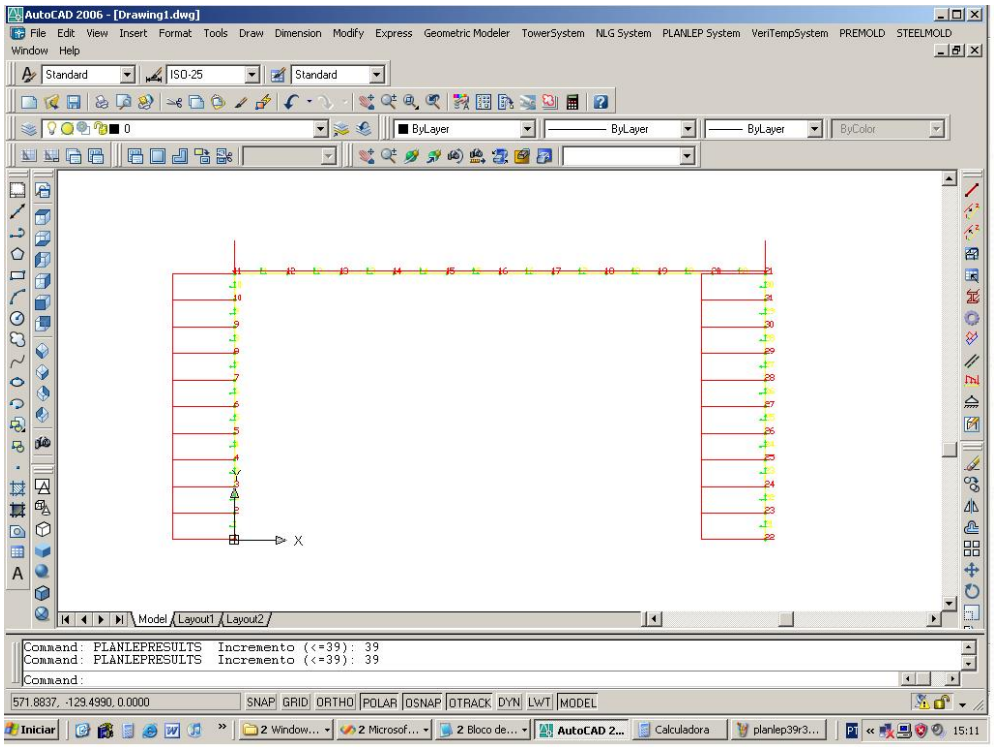

FIGURA 6.48 - Modelo refinado de pórtico de um pavimento e um vão com diagrama de esforços normais

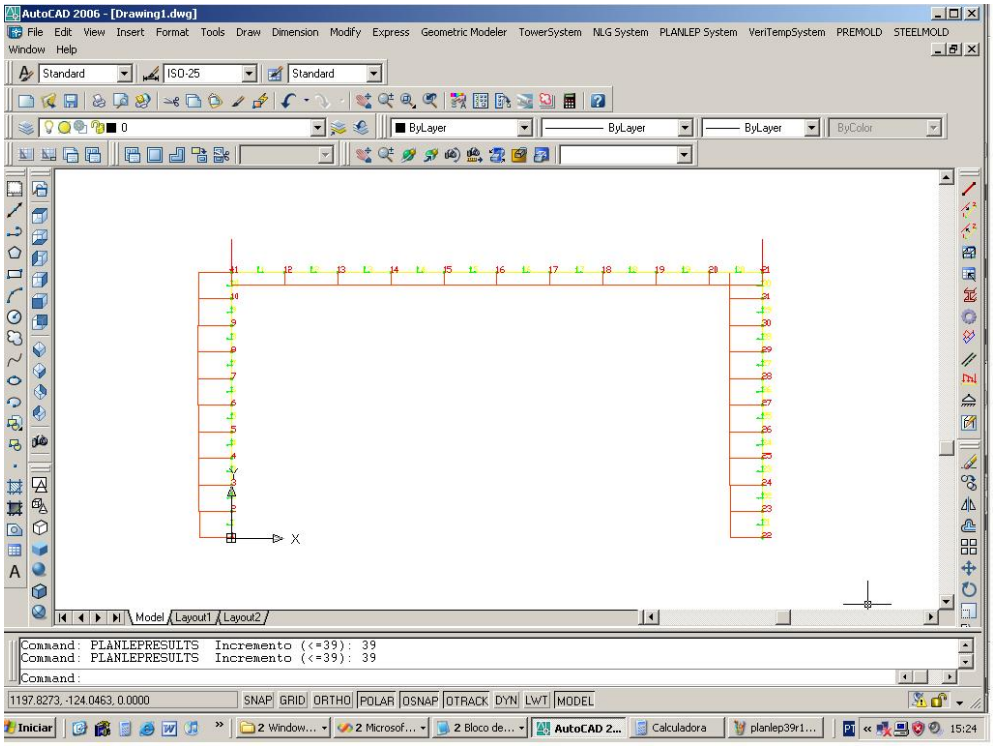

FIGURA 6.49 - Modelo refinado de pórtico de um pavimento e um vão com diagrama de esforços cortantes

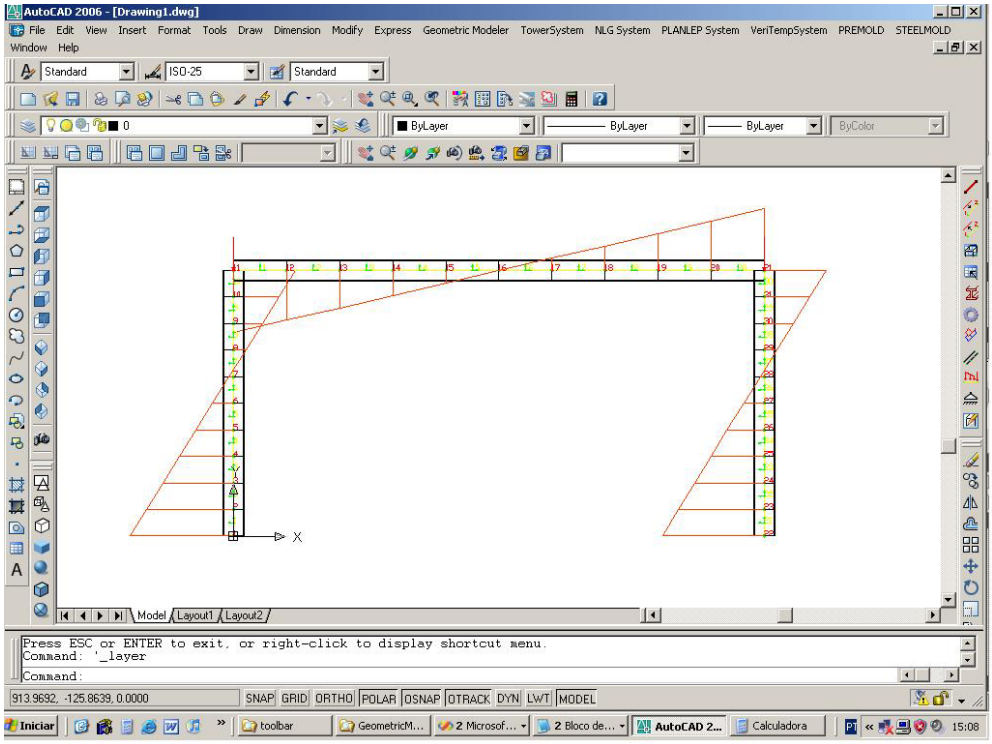

FIGURA 6.50 - Modelo refinado de pórtico de um pavimento e um vão com diagrama de momentos fletores

6.7 Aplicação VeriTemp[26]

A aplicação *VeriTemp* é um sistema de integração que faz a vinculação entre o sistema STEELMOLD, o sistema de análise PLANLEP e o sistema de dimensionamento VeriTemp (FIGURA 6.51), para verificação da resistência de perfis metálicos em temperatura ambiente e em situação de incêndio.

Nesta aplicação é possível modelar uma estrutura através do módulo PREMOLDED, utilizando os comandos do STELLMOLD, gerar um modelo discretizado através dos comandos do *Geometric Modeler,* exportar os dados do modelo discretizado para a análise via sistema PLANLEP, recuperar os dados da análise, tratá-los e exportá-los para o dimensionamento. O sistema de dimensionamento utilizado faz a verificação das barras do modelo individualmente, informando os elementos cuja inércia é insuficiente para resistir aos esforços solicitantes.

Para proceder à etapa de dimensionamento, é realizada a inserção automática dos valores de referência, obtidos da etapa de análise, no modelo geométrico (momentos fletores nos eixos X e Y, esforço cortante e esforço normal), obtendo os valores máximos por barra geométrica a partir do conjunto de elementos discretizados vinculados à mesma. Os dados da análise são formatados pela classe *VeriTempDimension* (FIGURA 6.52) gerando os dados de entrada para o sistema de dimensionamento. A classe *VeriTempDomain* configura os dados de dimensionamento

---

[26] SILVA, Alexandre Caram. Verificação do aumento de temperatura em elementos estruturais de aço. Versão 1.1.0. Verificação da resistência de perfis metálicos em temperatura ambiente e em situação de incêndio. Universidade Federal de Minas Gerais. Abril 2004

não presentes no modelo. Os dados de domínio para este sistema são: distância entre enrijecedores, coeficiente de correção do momento fletor, coeficiente de flambagem Kx e coeficiente de flambagem Ky.

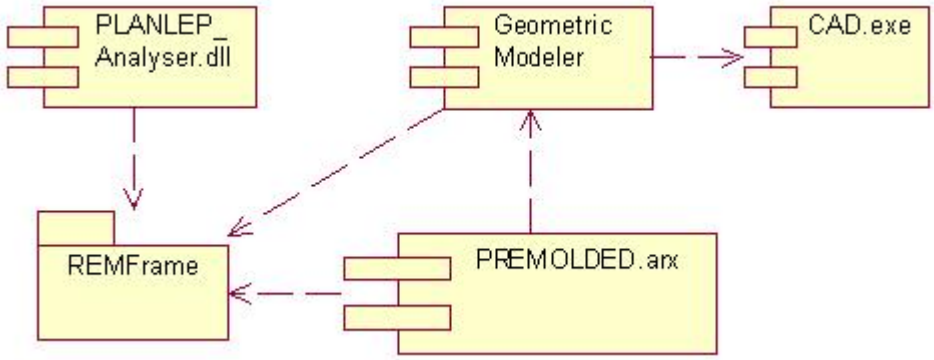

FIGURA 6.51 – Componentes da aplicação VeriTemp

A existência de duas classes de formatação de dados para transferência e de domínio é necessária para permitir a utilização dos dois sistemas externos. A instanciação dos objetos de formatação e de domínio é realizada pelas classes fábrica em tempo de execução. O comando responsável por acionar a etapa de análise antes de transferir a solicitação à classe de controle do sistema, através dos métodos de transferência de dados, repassa a esta última um objeto da classe *PLANLEPFactory,* configurando seu atributo fábrica. Após a etapa de análise, este atributo é alterado, em tempo de execução, pelo comando responsável por acionar o dimensionamento. Para efetuar a vinculação com o sistema de dimensionamento, o atributo fábrica passa representar um objeto da classe *VeriTempFactory*. Assim, quando a classe de controle solicita de sua fábrica os objetos de formatação e de domínio, recebe os objetos adequados à etapa de projeto em execução, repassando o controle de fluxo de dados destas etapas para os mesmos.

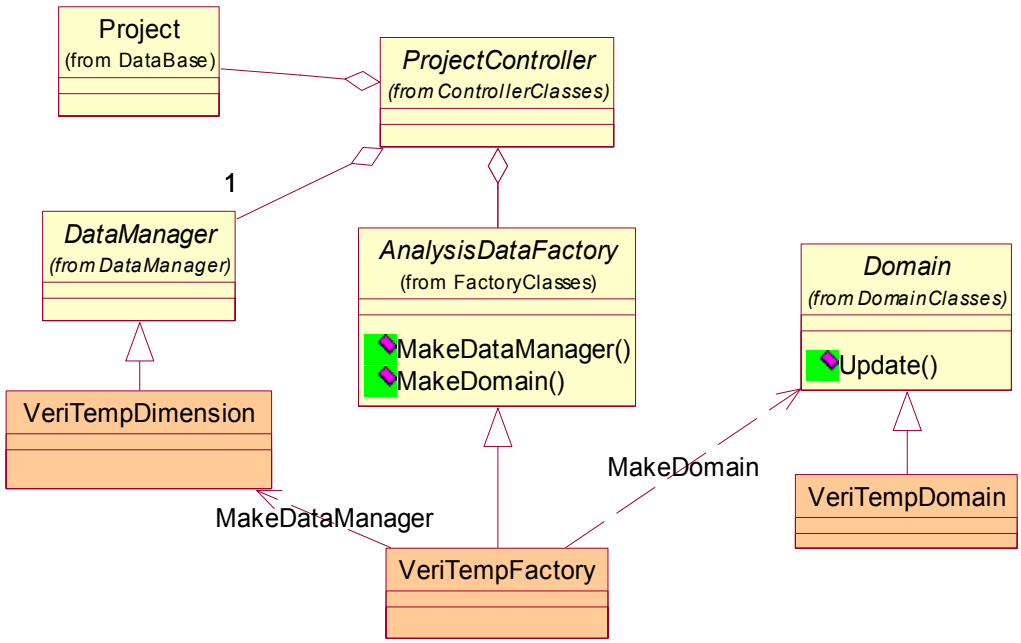

FIGURA 6.52 – Classes do componente VeriTemp

Após o dimensionamento, a aplicação realiza a leitura dos dados de saída do sistema, atualizando sua representação, mostrando através de diferenciação de cores, as barras cuja verificação indica não resistência aos esforços solicitantes, permitindo ao usuário que altere a seção transversal de tais elementos e realize novamente o dimensionamento.

7

CONCLUSÃO

Este capítulo apresenta as principais contribuições para o problema inicial proposto, as conclusões inferidas sobre os resultados obtidos e o alcance dos mesmos, além de sugestões para novos trabalhos.

7.1 Reflexões sobre as contribuições deste trabalho em relação aos objetivos e hipótese originais

Na busca pela automação do processo de projeto de estruturas reticuladas, deparamo-nos com alguns desafios básicos:

1. Abstrair um modelo capaz de representar todas as informações necessárias à correta definição da estrutura e necessárias aos vários agentes do processo;

2. Projetar estruturas capazes de armazenar e gerenciar estes dados, mantendo a consistência do modelo durante todo o processo;

3. Implementar a habilidade de transferir os dados, respondendo às necessidades dos agentes em cada etapa, de forma diferenciada.

4. Permitir uma visualização consistente do modelo, permitindo a representação de dados inseridos e de resultados obtidos nas diversas etapas;

5. Permitir a flexibilização dos sistemas utilizados ao longo do processo de acordo com o tipo de estrutura, o mecanismo de análise ou de dimensionamento, o mecanismo de detalhamento, etc.

A resposta a muitos destes desafios foi apresentada na forma do *framework* REMFrame, desenvolvido para permitir a integração das etapas do processo de projeto de estruturas reticuladas. Esta tese apresentou a arquitetura deste sistema e algumas aplicações que demonstraram a viabilidade da integração de diferentes sistemas em um processo de concepção estrutural, desde a modelagem até o dimensionamento. Não foram tratadas as integrações com as etapas de orçamento, de detalhamento e de manufatura devido às limitações de tempo. Porém, mesmo sem apresentar aplicações englobando todas as etapas do processo, acreditamos que o *framework* apresentado conseguiu responder satisfatoriamente a seu objetivo geral apresentado no Capítulo 1:

> "*O desenvolvimento de um sistema gerenciador de dados capaz de promover a integração de etapas do processo de projeto de estruturas reticuladas, visando contribuir para a melhoria da produtividade, da qualidade do processo e do produto.*"

Entende-se, também, que a hipótese levantada, também no Capítulo 1, foi confirmada e que, o *framework* desenvolvido foi capaz de responder aos desafios acima elencados, assim como aos seus objetivos específicos.

O primeiro objetivo específico elencado no Capítulo 1 foi assim definido: "Desenvolver o *framework* a partir da identificação clara de um domínio de aplicações dentro do processo de projeto de estruturas reticuladas. Apresentar uma abstração deste domínio com o uso dos conceitos de Engenharia de *Software* sob o paradigma da Orientação de Objetos, uso de padrões de projeto e de arquitetura, buscando-se obter um sistema robusto, confiável e re-aproveitável". Uma das formas de se avaliar a viabilidade do desenvolvimento de um *framework* para um domínio específico, é a análise de vários aplicativos semelhantes desenvolvidos dentro deste domínio, para a identificação de suas características principais, pontos em comum e fatores divergentes. Através do estudo realizado sobre vários aplicativos desenvolvidos para a modelagem de estruturas reticuladas, e a partir do estudo de sistemas, de ambientes e de um *framework* aplicados ao MEF, identificou-se a correlação entre os mesmos, sendo possível abstrair um núcleo de dados e de estruturas capazes de representar as características básicas tratadas pelos diversos sistemas. Este conjunto de dados, ou estruturas, foi modelado formando o núcleo do *framework,* o qual reúne as características que são independentes do tipo de estrutura modelada. Este núcleo é definido por classes, elementos, métodos e regras de relacionamento, tendo como base o um modelo de barras e nós. As características específicas requeridas pelos diferentes sistemas de análise e de dimensionamento utilizados, os diferentes tipos de inserção de dados no modelo ou mesmo as diferentes ligações entre os elementos de estruturas criadas, são tratadas em pontos de flexibilização do sistema, que devem ser desenvolvidos em cada aplicativo específico, constituindo os *hot spots* do *framework.*

Para responder de forma satisfatória aos desafios apresentados e à segunda parte do objetivo específico acima reproduzido, o *framework* foi apresentado através de uma

arquitetura desenhada a partir da adaptação de dois padrões de arquitetura e pela aplicação de diversos padrões de projeto documentados na literatura. Um dos problemas ainda hoje encontrados no desenvolvimento, na utilização ou na manutenção de *frameworks* é a falta de uma técnica de documentação normalizada para os mesmos. O uso de padrões no desenvolvimento do *framework* REMFrame permite que o mesmo seja conhecido a partir das decisões de projeto adotadas, dos padrões utilizados para modelar relacionamentos e funcionalidades e proporciona uma documentação para o mesmo auxiliando no entendimento do sistema para o desenvolvimento de aplicações.

O *framework* respondeu ao segundo objetivo específico enumerado: "Garantir a possibilidade de aplicar o *framework* no desenvolvimento de sistemas que possam utilizar diferentes plataformas gráficas para visualização dos dados tratados" através de uma importante característica: a independência entre dados e visualização. Esta característica representa dois importantes fatores de desenvolvimento: primeiro permite a flexibilização na escolha da plataforma gráfica a ser utilizada pelas aplicações e segundo, facilita a implementação das representações gráficas, uma vez que toda a manipulação de dados geométricos é realizada pelo *framework*, ficando a cargo das aplicações apenas a definição da visualização dos elementos. Desta forma um modelador pode apresentar versões para diferentes plataformas gráficas que utilizam a mesma base de manipulação e de gerenciamento de dados, apenas pela variação dos objetos de visualização e de características intrínsecas destas plataformas, como os mecanismos de obtenção de dados.

O objetivo específico de número três, que trata da validação do *framework* através do instanciamento de aplicações, foi alcançado pelas aplicações apresentadas no Capítulo 6. Estas aplicações foram desenvolvidas para utilizar sistemas pré-existentes e independentes, com formulação, aplicação, estrutura e utilização diferentes dos sistemas utilizados como referencial para o projeto da estrutura de dados e de funcionamento do *framework*. A escolha dos sistemas utilizados no desenvolvimento das aplicações se baseou, então, nas seguintes premissas:

- Independência com relação aos sistemas abordados na pesquisa bibliográfica;

- Facilidade de acesso aos arquivos executáveis destes programas, cedidos pelos seus autores;

- Acesso a documentações sobre as formatações requeridas pelos sistemas e suporte de seus desenvolvedores e autores;

- Existência de exemplos documentados na literatura, para validação das aplicações desenvolvidas.

O desenvolvimento dos modeladores gráficos: *Geometric Modeler*, novo PREMOLD e STEELMOLD, comparados aos modeladores anteriormente desenvolvidos e apresentados pelo grupo CADTEC, demonstraram o aumento de produtividade ao desenvolvimento de sistemas deste tipo proporcionado pelo uso do *framework*. Para avaliar ganho obtido em relação ao tempo de desenvolvimento destes sistemas podemos apresentar um cronograma simplificado das atividades assim distribuído:

- Pesquisa Bibliográfica e análise de domínio: 24 meses

- Projeto e implementação do *framework*: 18 meses

- Implementação das aplicações *Geometric Modeler* e *TowerSystem*, envolvento os primeiros testes e correções de tratamento de dados do núcleo do *framework* : 6 meses

- Implementação das demais aplicações: 6 meses

Temos, portanto, seis aplicações diferentes desenvolvidas em 12 meses, por um único desenvolvedor. Cabe aqui, porém, ressaltar que tanto o *framework* como as aplicações foram implementadas pelo mesmo desenvolvedor, ou seja, a curva de aprendizagem para a utilização do *framework* no desenvolvimento de aplicações, necessária a outros desenvolvedores, não foi avaliada nesta pesquisa.

Tomando então como base os testes de desenvolvimento, com a resalva da curva de aprendizagem do *framework*, podemos inferir algumas conclusões:

1. O desenvolvimento de sistemas de modelagem a partir de uma base pré-estabelecida permite, por exemplo, desenvolver sistemas especializados para modelar diferentes tipos de estruturas como: cimbramentos, estruturas de madeira, estruturas metálicas de perfis formados a frio, estruturas de treliças espaciais, etc. Com o uso do *framework*, o desenvolvimento destas aplicações concentra-se na definição de algumas seções transversais e no tratamento das ligações entre elementos através da instanciação dos *hot spots* de visualização e das classes concretas para modelar os diferentes tipos de ligações requeridos por cada tipo de estrutura. Uma vez que a arquitetura básica dos sistemas e todo o controle geral de fuxos são definidos pelo *framework,* o desenvolvimento dos diferentes modeladores pode concentrar-se nas soluções para as características específicas a serem tratadas. Desta forma é possível gerar soluções mais refinadas com menor custo de desenvolvimento que os modeladores criados como sistemas independentes. Estas características promovem, então, um aumento da produtividade do desenvolvimento de tais sistemas, incentivando a geração de modeladores para diferentes tipos de estruturas e, ainda, a verificação de sua aplicação em outras áreas semelhantes.

2. Outra importante característica do *framework* é a independência obtida entre dados e visualização o que representa dois importantes fatores de desenvolvimento: primeiro permite a flexibilização na escolha da plataforma gráfica a ser utilizada pelas aplicações e segundo, facilita a implementação das representações gráficas, uma vez que toda a manipulação de dados é realizada pelo *framework*, ficando a cargo das aplicações apenas a definição da visualização dos elementos. Desta forma um mesmo modelador pode apresentar versões desenvolvidas para diferentes plataformas gráficas, mas que utilizam uma mesma base de armazenamento de controle de fluxo de gerenciamento de dados.

3. Permitir o desenvolvimento de modeladores tridimensionais e sua integração a diferentes sistemas de análise representa uma importante contribuição para pesquisas acadêmicas. O desenvolvimento de sistemas gráficos representa um investimento de custos elevados e geralmente não é viável dentro do meio

acadêmico, se não for a atividade fim de uma pesquisa. Vários sistemas existentes nesse meio, desenvolvidos para a análise numérica aplicados a diversas teorias, não possuem pré e/ou pós-processadores. Muitas vezes são programas de alta qualidade numérica, mas que prescindem de uma representação gráfica para entrada de dados e para seus resultados. Nesse campo de aplicações, o *framework* surge como uma ferramenta interessante, ao permitir, através de um esforço computacional relativamente pequeno e simples, gerar aplicações que realizam o papel de pré e pós processadores para estes sistemas. Além disso, como é possível integrar em um mesmo processo diferentes sistemas externos, que desempenham uma determinada tarefa, esta integração permite, para uma mesma base de dados, obter os resultados de uma análise ou de um dimensionamento segundo diferentes abordagens, comparando resultados e desempenhos de uma maneira direta e precisa.

Assim, o *framework* REMFrame, ao ser aplicado sobre processos de projetos de estruturas reticuladas, abre caminho para a melhoria da produtividade e da competitividade deste importante setor da indústria de construção civil. O mesmo *framework* quando aplicado pela comunidade científica, como uma ferramenta para facilitar a interação com os sistemas numéricos desenvolvidos, contribui para a melhoria da qualidade do processo de pesquisa e, consequentemente, dos seus resultados, o que indiretamente acaba por se transformar também em melhoria da produtividade e da competitividade do processo de projeto de estruturas.

As contribuições deste trabalho podem então ser relacionadas a três áreas: aquisição e difusão de conhecimento, construção de *software* e produtividade de processos.

Uma importante contribuição deste trabalho está relacionada aos conhecimentos adquiridos e difundidos. Os estudos realizados ao longo deste trabalho, e sintetizados nesta tese, permitem a abertura de novos caminhos em busca de um reaproveitamento tanto de esforços quanto de conhecimentos anteriormente gerados. Uma base sintetizada de dados e de proposições permite a aplicação dos estudos aqui apresentados, referentes a conceitos, aplicações e desenvolvimentos de *frameworks*, aplicações de padrões e

análise comparativa de diversos trabalhos relacionados, em novas pesquisas, avançando sobre novos patamares até então não abordados.

Quanto ao desenvolvimento de sistemas computacionais podemos concluir que o desenvolvimento de um *framework* para o desenvolvimento de sistemas de integração para o processo de projeto de estruturas reticuladas é, não apenas viável, como bastante desejável, devido ao elevado reaproveitamento de esforços computacionais em um núcleo robusto e confiável, comum aos diversos sistemas utilizados ao longo do processo.

Outra importante contribuição desta pesquisa é possibilitar um aumento de produtividade para os processos de projeto de estruturas reticuladas e de desenvolvimento de sistemas computacionais. Esta produtividade pode estar vinculada tanto às atividades de mercado, quanto às atividades acadêmicas, pela capacidade de inserção das aplicações resultantes em ambos os setores.

7.2 Sugestões para futuros traballhos

Devido ao limite de tempo para o desenvolvimento da pesquisa, o foco da mesma foi a definição e a implementação da arquitetura do *framework* e no desenvolvimento de algumas aplicações em busca de validações da proposta apresentada. Não foi abordada uma otimização dos algoritmos utilizados para solução de problemas tais como: manipulações geométricas e a busca por dados em listas encadeadas ou em dicionários. Os algoritmos e os tipos de estruturas adotados para o armazenamento de dados visam responder de forma direta às necessidades do sistema, devendo ser otimizados para aumentar a eficiência do sistema, reduzindo tempo de execução e consumo de memória para armazenamentos.

Outro trabalho importante a ser desenvolvido é a vinculação do *framework* a bancos de dados para persistência de dados e para vinculação à etapa de orçamento e mesmo à etapa de planejamento que geralmente utilizam recursos de banco de dados.

A implementação das classes de tratamento de ligações é outra proposta para futuros trabalhos vinculados ao aprimoramento do *framework,* permitindo o desenvolvimento de aplicações que façam a vinculação com as etapas de dimensionamento e de detalhamento das ligações.

Outro fator a ser desenvolvido para aprimoramento do *framework* é a implementação de estruturas e de mecanismos que permitam desfazer e refazer operações sobre os dados gerenciados pelo núcleo *framework,* de forma consistente, considerando diferentes etapas de processamento. Estes mecanismos estão vinculados à viabilização do desenvovimento de comandos de edição nas aplicações desenvolvidas através do *framework*.

Propõe-se também o desenvolvimento de pesquisas que avaliem a vinculação do *framework*: a outros processos do empreendimento como o orçamento e o planejamento de etapas como transporte, montagem e execução das estruturas reticuladas.

Outra proposta para novos trabalhos é a avaliação da possibilidade de inserção de elementos planos no modelo de dados, assim como de estruturas que repesentem os elementos de dimensionamento (armações, soldas, inserts, etc.), no núcleo do *framework,* permitindo, através da inserção destes elementos, a integração das etapas do processo de projeto de estruturas, tomado sob um ponto de vista mais geral.

Um ponto importante a ser implementado, para a facilitação da utilização do *framework*, é o desenvolvimento de ferramentas do tipo wizards para acelerar e simpleficar o uso do *framework* por outros usuários. Estas ferramentas podem, por exemplo, auxiliar na construção de novos aplicativos através da instanciação automática de código para os *hot spots*.

REFERENCIAL BIBLIOGRÁFICO

ALMEIDA, Marcelo Lucas. ***Elementos Finitos Paramétricos implementados em Java***. 2005. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

AUTODESK Inc., ***ObjectARX for AutoCAD 2006 Help files***. Disponível em < http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=773180> Acesso em Abr 2006.

BALABRAM, A. ***Análise Limite de Vasos de Pressão via Programação Orientada a Objetos Aplicada ao Método dos Elementos Finitos. Modelamento/Analise/pós-processamento***. 2000. Dissertação de mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

BRAVO, C. M. A. A, DEVLOO, P. R. B., PAVANELLO, R. ***Sobre a implementação de um refinamento H-P***. Disponível em < http://www.fec.unicamp.br/~phil/> acesso em Jun 2007.

BURBECK, Steve. (1987) ***Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC).*** Disponível em <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html> Acesso em Ago. 2003

BUSCHMANN, F. MEUNIER, R. ROHNERT, H. SOMMERLAD, P. STAL, M. ***Pattern - Oriented Software Architecture A system of Patterns***. Chichester: John Wiley & Sons, 1996. 467p.

CARMO, Carla S. L. ***Automação de Detalhamento de Peças Padronizadas de Concreto Armado via CAD e POO***. 2001. Dissertação de Mestrado – Escola de Engenharia. UFMG, Belo Horizonte.

COPLIEN, James O. ***Software Design Patterns: Common Questions and Answers***. Software Production Researsh Department, AT&T Bell Laboratories. Disponível em < ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps > Acesso em Out 2003

DEVLOO, Philippe R. B. *PZ: **An object oriented environment for scientific programming***. Computer Methods in Applied Mechanics and Engineering 150, p133-153, 1997.

DEVLOO, Philippe R. B. ***Object Oriented Tools for Scientific Computing***. Faculdade de engenharia Civil, UNICAMP. May6, 1999. Disponível em < http://www.fec.unicamp.br/~phil/> acesso em Jun 2007

DUBOIS-PÈLERIN, Yves, ZIMMERMANN, Thomas, ***Object-Oriented finite element programming: III. An efficient implementation in C++***. Computer Methods in Applied Mechanics and Engineering 108, p 165-183, 1993.

DUBOIS-PÈLERIN, Yves, ZIMMERMANN, Thomas, BOMME, P. ***Object-Oriented finite element programming: II. A prototype program in Smalltalk***. Computer Methods in Applied Mechanics and Engineering 98, p 361-397, 1992.

EL DEBS, Mounir Khalil. ***Concreto pré-moldado: fundamentos e aplicações***. São Carlos: EESC-USP, 2000.

FAYAD, Mohamed E., SCHIMIDT, Douglas C. ***Object-Oriented Application Frameworks***. Communications of the ACM, Vol. 40, No. 10, p.32-38, Out 1997.

FAYAD, Mohamed E., SCHIMIDT, Douglas C. , JOHNSON, Ralph. E. ***Building Application Frameworks: Object-Oriented Foundations of Framework Design***. Wiley Computer Publishing, 1999. 664 pg.

FONSECA, Marcos T. ***Aplicação Orientada a Objetos Para Análise Fisicamente Não Linear com Modelos Reticuladas de Seções Transversais Compostas***. 2006. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

FRANCO J. R. Q., BARROS F. B., MALARD F. P., BALABRAM A. ***Object oriented programming applied to a finite element technique for a limit analysis of axisymmetrical pressure vessels***. Advances in Engineering Software, 37, p 195-204, 2006.

FRANCO, J. R. Q e LEITE, R. C. G. ***Tecnologia CAD.Aplicada à Automação do Processo de Estruturas Pré-moldadas de Concreto Armado: Modelagem e Detalhamento*** In: CILAMCE XXIV, 2003, Recife. *Anais Eletrônicos*...Ouro Preto: UFOP, 2003. 1 CD-ROM

FÜLLER, R.B., ***Synergetics: explorations in the geometry of thinking***. Macmillan Publishing Company, New York,1975.

FURLAN, José Davi. ***Modelagem de Objetos através da UML – The Unified Modeling Language*** – São Paulo: Makron Books, 1998. 329p

GAMMA, E., HELM R., JOHNSON R., VLISSIDES J. ***Padrões de Projeto: soluções reutilizáveis de software orientado a objetos***. Porto Alegre: Bookman, 2000. 363p.Trad.: Salgado, Luiz. A. Meirelles.

GONÇALVES, Marco Antônio Brugiolo. ***Geração de Malhas Bidimensionais de Elementos Finitos Baseada em Mapeamentos Transfinitos***, 2004. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

GONÇALVES, M. A. B. e PITANGUEIRA, Roque Luiz. ***Padrões de Projeto de Software para um gerador de malhas bidimensionais de elementos finitos.*** VI Simpósio Mineiro de Mecânica Computacional. Itajubá, 2004 Disponível em <http://www.dees.ufmg.br/insane/artigos.htm> Acesso em Mai 2007.

GOPAC/UFMG, GRUCAD/UFSC, LMAG/USP. ***BRFEM (Brazil Finite Elements): Architecture and Design***. Version 1.1, Maio 2003. Disponível em <http://www.ead.cpdee.ufmg.br/~renato/brfem> Acesso em Ago 2007.

GRECO M., GESUALDO F. A.R., VENTURINI W.S., CODA H.B. ***Non linear positional formulation for space truss analysis***. Finite Elements in Analysis and Design, N. 42 p. 1079-1086. Abr. 2006

HÜNTER A. ***Sistema de Modelagem CAD. Modelador 3D para Estruturas de Barras via CAD Engenharia/ Visualização/Banco de Dados.*** 1998. Dissertação de Mestrado - Escola de Engenharia, UFMG, Belo Horizonte.

LEITE, Regina Célia Guedes. ***Automação do Processo de Modelamento e Detalhamento de Estruturas Pré-Moldadas de Concreto Armado Via Tecnologia CAD e Programação Orientada a Objeto***. 2002. Dissertação de Mestrado - Escola de Engenharia, UFMG, Belo Horizonte.

LEITE, R. C. G. e FRANCO, J. R. Q. ***Arquitetura e Projeto de um Framework para a Modelagem de Estruturas Reticuladas Tridimensionais via Tecnologia CAD***. In: CILAMCE XXV, 2004, Recife. Anais Eletrônicos...Recife: UFPE, 2004. 1 CD-ROM

LEITE, R. C. G. e FRANCO, J. R. Q. ***Estudo da Aplicabilidade de Padrões de Arquitetura na Implementação de um Modelador CAD para Estruturas Espaciais Reticuladas***. In: CILAMCE XXVI, 2005, Guarapari. Anais Eletrônicos...Vitória: UFES, 2005. 1 CD-ROM

LEITE, R. C. G. e FRANCO, J. R. Q. ***Aplicação de um Framework na Modelagem, Análise e Pós-Processamento de Estruturas Reticuladas*** In: CILAMCE XXVII, 2006, Belém. Anais Eletrônicos...Belém: UFPA, 2006a. 1 CD-ROM

LEITE, R. C. G. e FRANCO, J. R. Q. ***Desenvolvimento de um Framework para Modelagem Via Tecnologia CAD de Estruturas Reticuladas Tridimensionais***. In: IX Encontro de Modelagem Computacional, 2006, Belo Horizonte. Anais Eletrônicos...Belo Horizonte: CEFET-MG, 2006b. 1 CD-ROM.

MAGALHÃES, Ana Liddy C. de Castro. ***Estudo, projeto e implementação de um modelador de sólidos voltado para aplicações em eletromagnetismo***. 2000. Tese de Doutorado – Departamento de Engenharia Elétrica, UFMG, Belo Horizonte.

MAGALHÃES, Paulo Henrique Vieira de. ***Automação do Processo de Modelamento e Detalhamento de Torres Via Tecnologia CAD***. 2002. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

MALARD, Fernando Poinho. ***Sistema de Modelagem CAD, Modelador 3D para Estruturas de Barras Via CAD. Arquitetura do Sistema/Banco de Dados/Visualização***. 1998. Dissertação de Mestrado - Escola de Engenharia, Universidade Federal de Minas Gerais.

MARKIEWICZ, M. E., LUCENA, C. J. P. ***Object Oriented Framework Development***. ACM Crossroads Student Magazine. 2001 Disponível em <http://www.acm.org/crossroads/xrds7-4/frameworks.html > Acesso em Ago 2003.

MATTSON, Michael, BOSCH, Jan, FAYAD, Mohamed E. ***Framework Integration Problems, Causes, Solutions***. Communications of the ACM, Vol.42, No. 10, p. 81-87, Outubro, 1999.

MCAULEY, C. ***Programming AutoCAD 2000® using ObjectARX™.*** Nova York: Thomsom Learning, 2000. 676p.

MICROSOFT Inc. ***Getting Started in .NET. Microsoft .NET***, 2004. Disponível em: <http://www.microsoft.com/net/> Acesso em Ago. 2004.

MICROSOFT Inc. ***Visual C++ Developer Center***, 2005. Disponível em: <http://msdn.microsoft.com/visualc/> Acesso em Mar 2005.

MARKIEWICZ, Marcus Eduardo, LUCENA, Carlos J.P. (2001), ***Object Oriented Framework Development***. ACM Crossroads Student Magazine - The ACM's First Eletronic Publication. Abril, 2001. P1-12. Disponível em <http://www.acm.org/crossroads/xrds7-4/frameworks.html >Consultada em Ago 2003.

MOREIRA, Renata N. ***Sistema Gráfico Iterativo para Ensino de Análise Estrutural Através do Método dos Elementos Finitos***. 2006. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

PAULA FILHO, Wilson de Pádua (2001). ***Engenharia de Software: fundamentos, métodos e padrões***. LTC Editora, 2001.

PENNA, Samuel Silva, ***Pós-Processador Para Modelos Bidimensionais Não Lineares Do Método Dos Elementos Finitos***, 2007. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

PZ – ***An h-p adaptive finite element environment PZ*** Disponível em < http://www.fec.unicamp.br/~phil/> . Acesso em Nov 2005.

QUATRANI, Terry (1998). ***Visual Modeling with Rational Rose and UML,*** Massachusetts: Addison-Wesley, 1998. 222p.

ROBERTS, Don e JOHNSON, Ralph. ***Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks***. Disponível em <http://st-www.cs.uiuc.edu/users/droberts/evolve.html >

SCHIMIDT, Douglas C., FAYAD, Mohamed E., Johnson, RALPH. E. ***Software Patterns***. Communications of the ACM, Vol.39, No. 10, p.36-39, Outubro, 1996.

SHALLOWAY, Alan e TROTT, J. R. ***Explicando padrões de projeto: uma nova perspectiva em projeto orientado a objeto***.Porto Alegre: Bookman, 2004. 328 p.

.

SILVA, Maria A. Covelo e SOUZA, Roberto de. ***Gestão do Processo de Projeto de Edificações.*** São Paulo: O Nome da Rosa, 2003.

SILVA R.G.L. e LAVALL A.C.C. ***Avaliação dos Efeitos de 2ª. Ordem em Edifícios altos de aço utilizando o método aproximado de amplificação dos momentos em comparação com um método de análise rigorosa***. In: CILAMCE XXVI, 2005, Guarapari. Anais Eletrônicos...Vitória: UFES, 2005. 1 CD-ROM

SILVA R.G.L. ***Avaliação dos Efeitos de 2ª. Ordem em Edifícios de aço utilizando métodos aproximados e análise rigorosa***. 2004. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

SOARES, Bruno Cesarino. ***Técnica de Elementos Finitos para Análise Limite de Pórticos Planos Associada à Automação do Processo de Modelagem 3D de Estruturas Reticuladas Via CAD***. 2006. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

SOUZA, R. ***Sistema de gestão da qualidade para empresas construtoras***. São Paulo: PINI, 1995. p. 127-147

USIMINAS MECÂNICA. ***Lista de Perfis Soldados***. Disponível em: <http://www.umsa.com.br/ext/perfis_metalicos/soldados/xls/tabela%20soldado.xls> Acesso em 24 mai 2007.

VARALLA, Ruy. ***Planejamento e controle de Obras***. Coleção Primeiros Passos da Qualidade no Canteiro de Obras, Ed. O Nome da Rosa. São Paulo, 2003. p. 9-41

ZIMMERMANN, Thomas, DUBOIS-PÈLERIN, Yves e Bomme, Patrícia. ***Object-Oriented Finite Element Programming: I. Governing Principles***. Computer Methods in Applied Mechanics and Engineering 98(1992) 291-303

OBRAS CONSULTADAS

BATTISTI, Júlio. ***O Modelo Relacional de Dados – Parte 5***. Dsiponível em http://www.juliobattisti.com.br/artigos/office/modelorelacional_p5.asp Acesso em Mar 2005

BERG, Mark de; KREVELD, Marc van; OVERMARS Mark; SCHWARZKOPF Otfried**.** ***Computational Geometry: Algorithms and Applications***, Second Edition, Springer Verlag, 2000. 367pg

CULP, Timothy R. (1999). ***Industrial Strength Pluggable Factories***. Journal of C++ Report. Disponível em <http://www.adtmag.com/joop/crarticle.asp?ID=1520> Acesso em Jan 2004

ECKEL, Bruce, ***Thinking in C++ - vols 1 & 2***, 2nd Edition, Prentice Hall, 1999. Disponível em:<http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html> Acesso em mai 2002.

FRANÇA, Júnia L. e Vasconcelos, Ana C. de. Colaboração: MAGALHÃES, M. H.de A. e BORGES S. M. ***Manual para normalização de publicações técnico-científicas.*** 7. ed. – Belo Horizonte: Ed. UFMG, 2004 242p.

FUINA, Jamile Salim. ***Métodos de Controle de Deformações para análise não linear de estruturas*****,** 2004. Dissertação de Mestrado – Escola de Engenharia, UFMG, Belo Horizonte.

GEHBAUER, Fritz. ***Planejamento e Gestão de Obras: um resultado prático da cooperação técnica Brasil-Alemanha***. Curitiba: CEFET-PR, 2002. p. 39-57

MCILRATH, Michael (1994) ***Technology CAD Framework***. 1994. Disponível em <http://www-mtl.mit.edu/MTL/report94/MAN/tcadfw.html > Acesso em Nov. 2003

PACE J. A. D., TRILNIK, M. R. C., FAYAD, M. E. ***Accomplishing Adaptability in Simulation Frameworks: the Bubble Approach***. Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International Volume , Issue , 2001 Page(s):437 - 444

SILVA E.J., MESQUITA R.C., SALDANHA R.R. e PALMEIRA P.F.M., ***Object oriented finite element program for electromagnetic field computations***, Proc. 9th Conf. on the Computation of Electromagnetic Fields (COMPUMAG'93). Miami, FL. USA, IEEE Trans. Magnet. 30(5) pt 2 (Sept. 1994).

STROUSTRUP, Bjarne (1997) ***The C++ Programming Language,*** third edition, Addison-Wesley, 1997.

THOMAZ, Ércio. ***Tecnologia, Gerenciamento e Qualidade na Construção***. São Paulo: PINI, 2001. p. 93-100