UNIVERSIDADE FEDERAL DE MINAS GERAIS
Escola de Engenharia
Departamento de Engenharia de Estruturas
Curso de Pós-Graduação em Engenharia de Estruturas

GERAÇÃO DE MALHAS BIDIMENSIONAIS DE ELEMENTOS FINITOS BASEADA EM MAPEAMENTOS TRANSFINITOS

Marco Antônio Brugiolo Gonçalves

Dissertação apresentada ao curso de Pós-Graduação em Engenharia de Estruturas da UNIVERSIDADE FEDERAL DE MINAS GERAIS, como parte dos requisitos para obtenção do título de MESTRE EM ENGENHARIA DE ESTRUTURAS.

Orientador: Prof. Dr. Roque Luiz da Silva Pitangueira

Belo Horizonte
Novembro de 2004

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

"GERAÇÃO DE MALHAS BIDIMENSIONAIS DE ELEMENTOS FINITOS BASEADA EM MAPEAMENTOS TRANSFINITOS"

Marco Antônio Brugiolo Gonçalves

Dissertação 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 "Mestre em Engenharia de Estruturas".

Comissão Examinadora:

_____
Prof. Dr. Roque Luiz da Silva Pitangueira
DEES - UFMG - (Orientador)

_____
Prof. Dr. Renato Cardoso Mesquita
DEE - UFMG

_____
Prof. Dr. Felicio Bruzzi Barros
CEFET - MG

Belo Horizonte, 05 de novembro de 2004

Todo conhecimento começa com o sonho. O conhecimento nada mais é que a aventura pelo mar desconhecido, em busca da terra sonhada.

*(*Rubens Alves*)*

O papel de uma teoria científica não é o de fornecer uma solução tão geral dos problemas que se torne irrefutável à experiência, mas, ao contrário, o de abrir novos caminhos sobre os quais se reencontrarão, cedo ou tarde, novos obstáculos fecundos.

*(*Jean Piaget*)*

Nada é permanente, exceto as mudanças.

*(*Heráclito*)*

Dedico este trabalho aos Meus Pais.

Índice

Lista de Tabelas

Lista de Figuras

Resumo

Esta dissertação é parte do desenvolvimento de um ambiente computacional para disponibilizar diferentes modelos discretos de análise estrutural. A parte do sistema tratada neste trabalho é o pré-processador, que consiste numa aplicação gráfica interativa, implementada na linguagem Java, utilizando o paradigma de programação orientada a objetos, para geração de malhas bidimensionais de elementos finitos.

Este trabalho documenta e destaca a importância das fases de Análise, Projeto e Implementação Orientados a Objetos. Padrões de projeto de *software* reconhecidamente eficientes são adotados na implementação desta aplicação. Visando separar o modelo de sua representação, a implementação é baseada na metáfora de programação denominada *Model-View-Controller* (*MVC*). Tal enfoque permite que o controle da geração da malha, através de interação com usuário, e a visualização da mesma sejam implementados independentemente do modelo adotado, minimizando as tarefas de manutenção e expansão do sistema. O *MVC* propicia também o aperfeiçoamento gradual da aplicação através de mudança de plataforma, criação de diversas vistas sincronizadas com o modelo e substituição ou atualização das diversas vistas.

No primeiro modelo disponibilizado pela aplicação, as malhas são geradas através de mapeamentos transfinitos. A geometria do modelo é representada através de sub-regiões definidas por curvas do contorno que, por sua vez, são constituídas por um conjunto de primitivas (pontos, segmentos de retas, curvas quadráticas e cúbicas). Os tipos de mapeamentos disponibilizados são `lofting`, `bilinear` e `trilinear` e os elementos finitos podem ser serendípticos ou lagrangeanos, triangulares ou quadrilaterais. Uma vez gerada a malha, permite-se a prescrição de atributos de carga, apoio, material, dentre outros, e o modelo de elementos finitos gerado pode ser persistido em arquivo `XML` ou `Objeto Java` para posterior utilização em programas de análise.

Abstract

This master´s thesis is part of development of a computational environment to make available different discreet models of structural analysis. Part of the system, related in this work, is the pre-processor that consists in an interactive graphic application, implemented in Java language, using the object-oriented paradigm for two-dimensional finite elements mesh-generation.

This work documents and focus on object-oriented Analysis, Project and Implementation. Admittedly efficient software design patterns are adopted in the implementation of the application. To separate the model from its representation, the implementation is based on *Model-View-Controller* (*MVC*) programming metaphor. Such metaphor allows the control of mesh generation, through user interaction, and its visualization be implemented independently of the adopted model, minimizing the system maintenance and expansion tasks. Besides, the *MVC* lets the gradual improvement of application through platform change, creation of several views synchronized with the model and replacement or update of several views.

In the first model made available by the application, the meshes are created through transfinite mappings. The geometry of the model is represented through sub-regions that are defined by boundary curves. These curves are formed by a set of primitives like points, lines, quadratic and cubic curves. The available mappings are lofting, bilinear and trilinear. Otherwise, the finite elements can be serendipity or lagrangean, triangular and quadrilateral. Once created the mesh, it allows the attribution of load, support, material, among others. Finally, the model of created finite elements can be saved either XML or Java object file for posterior application in analysis programs.

Agradecimentos

A *DEUS* que sempre ilumina meus caminhos, me acompanha em todos os momentos e compreende minhas atitudes, e reconhecendo meus esforços me concedeu mais uma vitória: chegar ao final dessa longa jornada.

Aos *meus pais* que compreenderam minha ausência ao longo do período em que este sonho estava em construção, que sempre me apoiaram e acreditaram na realização desta conquista.

À *Jamile* que me ajudou a concretizar este sonho, seu auxílio foi fundamental, em todas as etapas deste trabalho.

Aos *meus irmãos* que me auxiliaram, incentivaram e ouviram muitos pedidos, geralmente desacompanhados das palavaras *por favor* e *obrigado*.

Aos *meus padrinhos, tios, primos e amigos de BH* que me receberam muito bem, ofereceram suas casas, auxílio, companhia, amizade, enfim, tornaram esse período nessa cidade mais agradável, prazeroso e humano.

Aos *demais membros da minha família* que torceram por mim, me incentivaram e sempre cobraram minha presença. Essa vitória foi obtida a custo de muita saudade.

Aos *amigos de JF* com os quais deixei de tomar muita cerveja e curtir algumas ressacas.

Ao meu orientador, *Roque Luiz da Silva Pitangueira*, pela orientação exemplar em todos os momentos deste trabalho.

Aos *professores* e *funcionários* do Departamento de Engenharia de Estruturas da $UFMG$ pela disponibilidade e atenção em todos os momentos.

A todos aqueles que de alguma forma contribuíram para a realização deste trabalho.

Ao *CNPQ* pelo apoio financeiro.

Capítulo 1

INTRODUÇÃO

Costuma-se dividir o processo de análise estrutural em três etapas (Soriano & Lima 1999). Orientando-se por experiência de projeto, o problema de meio contínuo da estrutura real é substituído por um modelo matemático utilizando-se hipóteses simplificadoras. Tal modelo matemático é expresso por equações diferenciais (ordinárias ou parciais) cujas soluções, ditas soluções analíticas, são conhecidas apenas em alguns poucos casos simples. Para superar as limitações de tais soluções, adota-se um modelo numérico aproximado, dito modelo discreto. Nos modelos discretos, as equações são algébricas e as grandezas são determinadas em um número finito de pontos, diferentemente das soluções analíticas cujas equações diferenciais, quando resolvidas, permitem avaliar as grandezas em um número infinito de pontos. Dentre os métodos discretos mais utilizados destacam-se o Método dos Elementos Finitos (MEF) e o Método de Elementos de Contorno (MEC). A pesquisa na área de métodos numéricos e computacionais para os referidos modelos discretos procura um aprimoramento das hipóteses simplificadoras dos mesmos. Isto é feito de forma a ampliar complexidades a partir dos conceitos já consolidados. Entretanto, observa-se sempre um recomeço do processo ao se recriarem as ferramentas relativas às tecnologias dominadas. Um exemplo ilustrativo deste fato é a (re)implementação computacional de algoritmos de solução de sistemas de equações algébricas lineares, toda vez que os mesmos são usados como parte do processo de aprimoramento de determinado modelo discreto.

Ao longo do tempo, algumas iniciativas de desenvolvimento de *software* pela comunidade acadêmica resultaram em produtos dependentes de sistema operacional, pouco amigáveis, escritos em linguagens de programação não apropriadas, de expansão, distribuição e manutenção difíceis, desenvolvidos por equipes fechadas, com documentação deficiente, entre outras limitações. Tais fracassos podem ser creditados à falta de disposição da comunidade em se apropriar das tecnologias emergentes ou mesmo à inexistência das mesmas.

Esta constatação confronta-se com o surgimento e aprimoramento de recursos para desenvolvimento de *software*, como programação orientada a objetos, linguagem Java, XML (eXtensible Markup Language), padrões de projeto de *software*, entre outras. Soluções estas que permitem o desenvolvimento de sistemas computacionais segmentados, amigáveis a mudanças e escaláveis em complexidade (Alvim 2003).

Portanto, o desafio de desenvolver sistemas utilizando estes recursos é condição obrigatória para aprimoramento da agilidade e criatividade da pesquisa na área.

1.1 Objetivos Gerais

A utilização de modelos discretos de análise estrutural compreende três etapas principais inter-relacionadas: (1)criação do modelo, (2)montagem e resolução do modelo e (3)avaliação de resultados. Na criação do modelo, o analista informa as hipóteses simplificadoras relativas à geometria, material, carregamento e condições de contorno e estas são representadas por meio de entidades matemáticas apropriadas, gerando assim o que se denomina malha e os atributos do modelo. Na etapa de montagem e resolução do modelo, combinam-se as informações matematicamente representadas, de modo a produzir equações algébricas que, quando solucionadas, permitem obter as diversas grandezas. Na avaliação de resultados, o analista faz uma análise crítica e verifica a adequação dos mesmos ao problema em estudo.

Para disponibilização deste processo em computadores, normalmentente a referida divisão é adotada através de programas de pré-processamento (para a criação dos modelos com recursos gráficos interativos), processamento (para a montagem e resolução numérica do modelo) e pós-processamento (para visualização gráfica de resultados).

As possibilidades que os recursos tecnológicos para desenvolvimento de *software* oferecem para cada uma das três etapas constituem amplo campo de pesquisa na área de métodos numéricos e computacionais aplicados à engenharia.

O domínio destes recursos e a aplicação dos mesmos no aprimoramento progressivo dos modelos, sem ter que recomeçar o processo a cada novo aperfeiçoamento, requer um ambiente computacional segmentado, amigável a mudanças e escalável em complexidade.

O projeto `INSANE` ("INteractive Structural ANalysis Environment") objetiva desenvolver um sistema computacional com estas características. Para isto, o ambiente possui três grandes segmentos (pré-processador, processador e pós-processador).

Os pré e pós-processadores são aplicações gráficas interativas, implementadas na linguagem Java, que disponibilizam recursos diversos para diferentes modelos discretos. O processador é uma aplicação, também implementada em Java, que representa o núcleo numérico do sistema. Este núcleo é responsável pela obtenção dos resultados de diferentes modelos discretos de análise estrutural. A persistência dos dados compartilhados pelas três aplicações é alcançada através de uma interface baseada em arquivo(s) XML e/ou objetos Java.

Cada um dos três segmentos da aplicação é implementado segundo o paradigma de programação orientada a objetos (POO), adotando-se uma arquitetura em camadas e padrões de projeto de *software* apropriados.

1.2 Objetivos Específicos

Quando o Método dos Elementos Finitos surgiu, cabia ao engenheiro analista a tediosa tarefa de descrever as malhas em arquivos texto diretamente para os programas de análise disponíveis na época.

Atualmente, são utilizados programas gráficos de pré e pós-processamento para geração das malhas e visualização dos resultados. A descrição da malha através de arquivos texto foi substituída pela definição do modelo dentro de um ambiente gráfico interativo que permite fornecer as informações necessárias ao programa de análise. A utilização de programas gráficos de pré-processamento proporcionou grande redução no esforço requerido na definição

do modelo e conferiu maior precisão e qualidade aos dados que descrevem o problema. A análise dos resultados feita através da consulta a inúmeras folhas de listagem foi substituída pela visualização gráfica, onde os resultados podem ser apresentados sob diversos modos de representação gráfica (Miranda 1999).

Apesar de toda a sofisticação conferida aos pré-processadores existentes atualmente, estes estão em contínua evolução devida às mudanças ocasionadas pelos aperfeiçoamentos de *software* e *hardware*.

A dissertação de mestrado que aqui se apresenta refere-se ao segmento pré-processador do `INSANE`. Trata-se de uma aplicação gráfica interativa para geração de malhas de modelos discretos.

No primeiro modelo disponibilizado pela aplicação, malhas bidimensionais de elementos finitos são geradas através de mapeamentos.

Visando separar o modelo de sua representação, a implementação é baseada no padrão de projeto Model-View-Controller (Pietro 2001). A utilização desta metáfora de programação permite que o controle da geração da malha, através de interação com o usuário, e a visualização da mesma sejam implementados independentemente do modelo adotado, minimizando as tarefas de manutenção e expansão da aplicação.

1.3 Organização do Texto

Esta dissertação está apresentada em 6 capítulos.

No capítulo 2 discute-se técnicas de geração de malhas em domínios bidimensionais, destacando-se aquelas baseadas em mapeamentos. Neste mesmo capítulo elege-se a geração de malhas baseada em *Mapeamentos Transfinitos* para implementação, apontando-se suas principais características, vantagens e desvantagens.

O capítulo 3 discute as tecnologias de desenvolvimento de *software* empregadas. Após apresentação dos principais conceitos do paradigma de programação orientada a objetos, discute-se os padrões de projetos de *software* adotados e justifica-se a escolha de Java como linguagem de implementação.

No capítulo 4 procede-se a análise e o projeto orientados a objetos do gerador de malhas. A adoção da arquitetura em camadas e sua ligação com padrões de projeto é discutida. A seguir, os requisitos, funções, *casos de uso* e *modelo conceitual* do sistema são apresentados. Finalmente, as classes criadas e as associações entre elas são apresentadas utilizando diagramas *UML* (*Unified Modeling Language*) apropriados.

No capítulo 5 são apresentados vários exemplos de malhas de elementos finitos obtidas com o programa. Estes exemplos apresentam as principais características dos mapeamentos transfinitos implementados, bem como as possibilidades que os mesmos oferecem para a geração de malhas.

Finalmente, no capítulo 6, os principais avanços do `INSANE`, obtidos com esta dissertação são apresentados. Apresenta-se também várias sugestões para o aprimoramento do sistema.

O aplicativo desenvolvido conta com uma documentação no formato HTML, segundo o padrão sugerido pela Sun Microsystems. Esta documentação é complementada por dois manuais: o Manual de Desenvolvimento (Brugiolo & Pitangueira 2004*a*) e o Manual de Utilização do Sistema (Brugiolo & Pitangueira 2004*b*), disponíveis em `www.dees.ufmg.br/insane`.

O Manual de Desenvolvimento detalha a implementação propriamente dita do sistema. As API´s (Aplication Program Interfaces) Java utilizadas para a criação da interface com o usuário, visualização gráfica e manipulação das estruturas de dados são apresentadas no contexto do projeto orientado a objetos estabelecido.

O funcionamento do pré-processador é detalhado no Manual de Utilização do Sistema. Através de exemplos, os recursos disponibilizados para geração de malhas, desde a definição da geometria até a persistência de dados para a análise via Método dos Elementos Finitos, são apresentados.

Capítulo 2

GERAÇÃO DE MALHAS

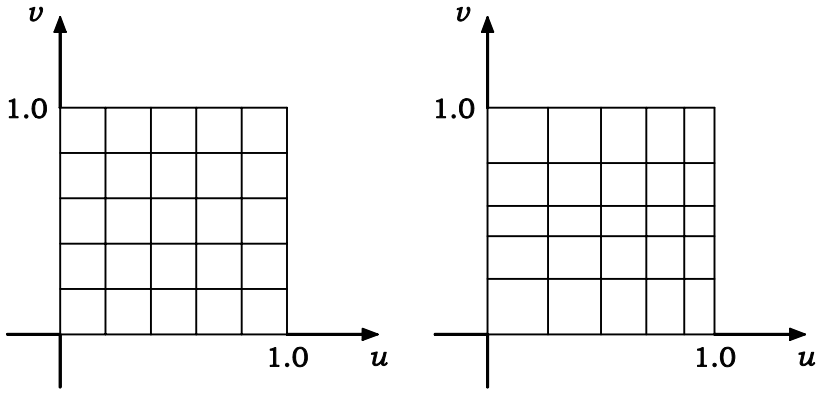
2.1 Técnicas de Geração de Malhas

Conforme discutido em Fonseca (1989), os métodos de geração de malhas em domínios bidimensionais podem ser classificados em diretos ou algébricos e indiretos ou de equações diferenciais.

Os métodos diretos ou algébricos são assim chamados porque geram uma malha sobre o domínio, baseados em algum algoritmo algébrico definido.

Os métodos diretos podem ser subdivididos em geração de malhas em domínios elementares, geração de malhas por transformação de coordenadas, mapeamentos conformes, mapeamentos isoparamétricos, mapeamentos transfinitos e decomposição de domínio.

A geração de malhas em domínios elementares (figura 2.1) não constitui propriamente



(a) (b)

Figura 2.1: Malhas em domínios elementares

um método de geração de malhas. Entretanto, as malhas em domínios elementares são úteis para a geração de malhas em domínios mais complexos usando-se técnicas de transformação de coordenadas ou de mapeamento. No caso de se desejar obter uma maior concentração de pontos numa determinada região pode-se usar uma função de densidade de malha (figura 2.1b).

A geração de malhas por transformação de coordenadas é o tipo mais simples de mapeamento e consiste em usar uma transformação de coordenadas que mapeia um domínio elementar no domínio real desejado (figura 2.2). A desvantagem desse método é que torna-se necessário determinar a transformação de coordenadas que faça o mapeamento do domínio elementar no domínio real, o que nem sempre é simples ou mesmo possível.

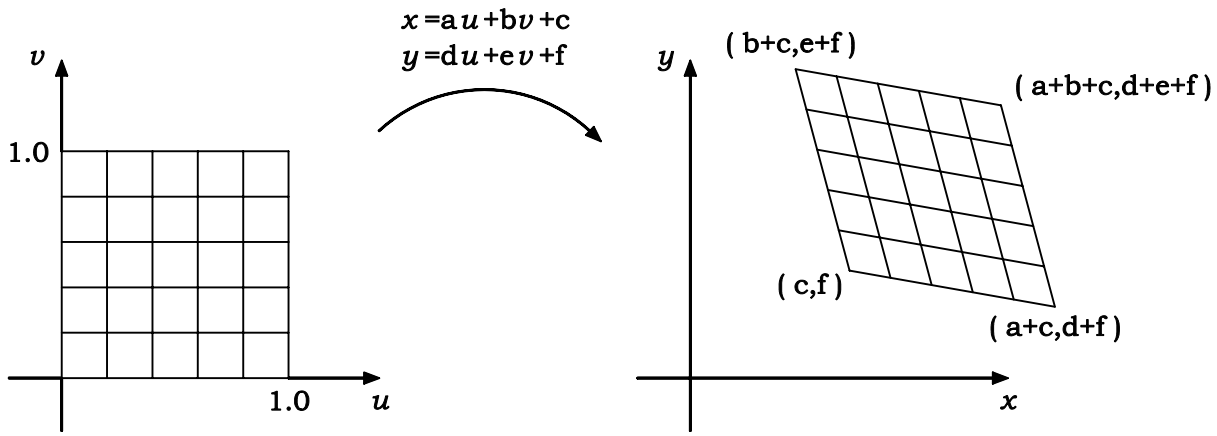


Figura 2.2: Malhas geradas por transformação de coordenadas

O mapeamento conforme consiste em associar os pontos dos domínios elementar e real a números complexos (Kreyszig 1993). Desta forma, o mapeamento procurado torna-se uma função de variável complexa que mapeia o domínio elementar no domínio real desejado (figura 2.3). Entre as desvantagens desse método está o fato de que as características da malha são definidas automaticamente dificultando a produção de uma malha adequada ao problema a ser solucionado. Esse método também exige que se determine a transformação de coordenadas que faça o mapeamento do domínio elementar no domínio real, além disso podem surgir dilatações ou adensamentos indesejáveis na malha.

O mapeamento isoparamétrico consiste em obter os valores das coordenadas dos pontos do domínio a partir de valores especificados no contorno através do uso de funções de interpolação

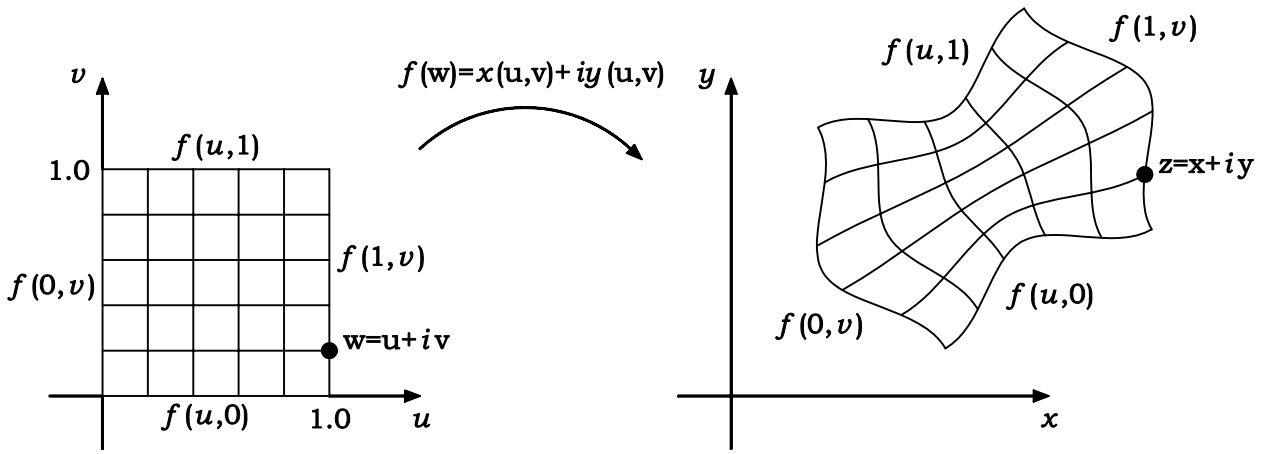


Figura 2.3: Mapeamento conforme

(figura 2.4). Nesse processo o contorno do domínio não é especificado, ele é aproximado por funções de interpolação que passam por pontos especificados. Essas mesmas funções de interpolação são usadas pra mapear a malha no domínio. Nesse tipo de mapeamento é possível controlar a densidade de malha respeitando certos limites, através do posicionamento adequado dos nós dos centros dos lados.

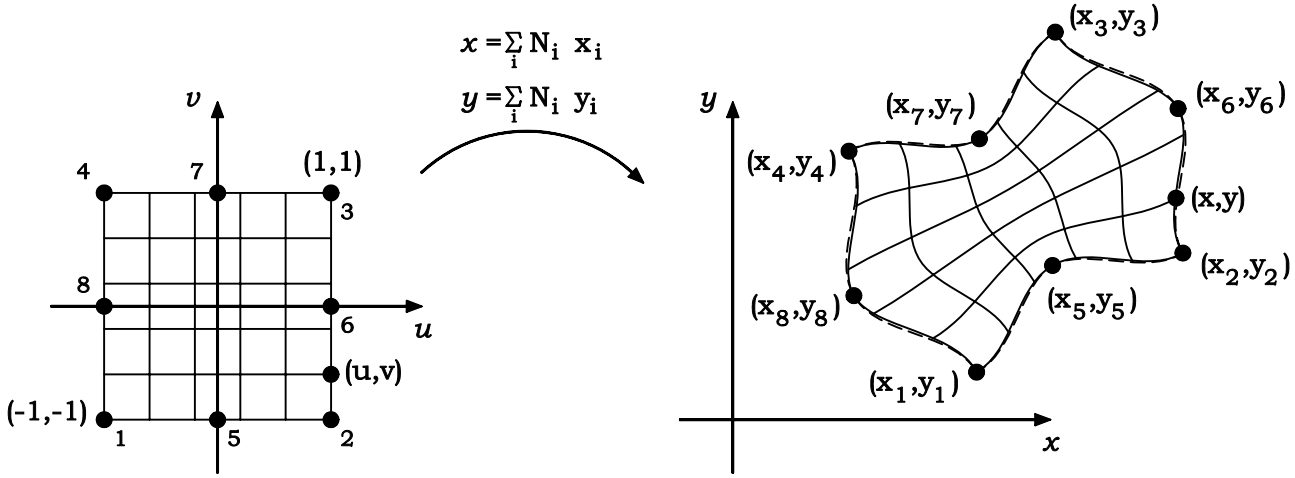


Figura 2.4: Mapeamento isoparamétrico

O mapeamento transfinito estabelece sistemas de coordenadas curvilíneas definidos pelo contorno de domínios arbitrários (figura 2.5). Este método descreve uma superfície aproximada que coincide com a superfície real ou idealizada em um número não enumerável de pontos, propriedade que deu origem ao nome mapeamento transfinito. É capaz de modelar o contorno de superfícies sem a introdução de nenhum erro na geometria do mesmo.

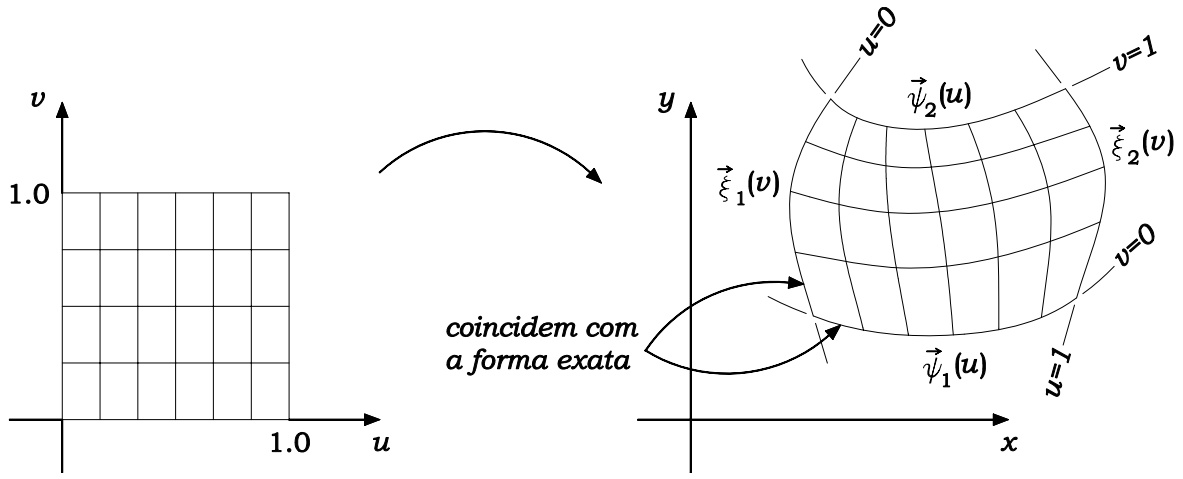


Figura 2.5: Mapeamento transfinito

Os métodos de decomposição de domínio (figura 2.6) são baseados em técnicas de decomposição espacial, como árvore quartenária (quadtree), avanço de fronteira e o método de Delauney. Diversos trabalhos têm sido desenvolvidos nesta área como discutido por Miranda (1999).

A geração de malhas através de decomposição de domínio é uma alternativa relevante nos casos de domínios com formas muito complexas e em casos onde se deseja variar drasticamente a densidade de elementos ao longo do domínio.

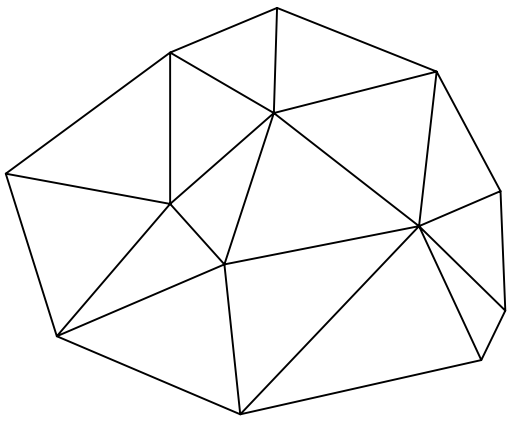


Figura 2.6: Decomposição de domínio

Nos métodos indiretos, a geração da malha envolve a solução de uma equação diferencial a fim de obter um sistema de coordenadas curvilíneas, onde as linhas de coordenadas constantes são as linhas da malha (figura 2.7). Estes métodos são mais onerosos que a maioria dos métodos algébricos e, portanto, não são apropriados para sistemas interativos de geração de

malhas devido ao elevado tempo de resposta.

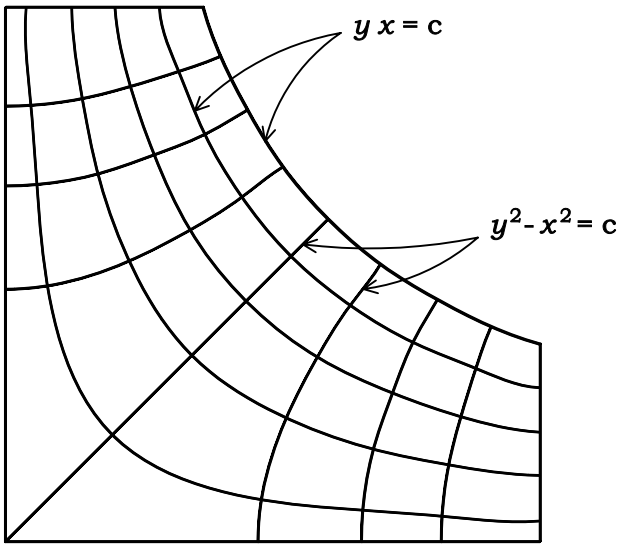


Figura 2.7: Malha gerada por métodos indiretos

Num sistema gráfico interativo o processo de entrada de dados de geometria e topologia fica muito mais fácil e flexível. Entretanto, para servir a um sistema interativo um algoritmo deve fornecer respostas rápidas e eficientes. Portanto, para atender a essas exigências, os métodos de mapeamentos mais adequados são os de mapeamento isoparamétricos ou transfinitos (Fonseca 1989).

2.2 Mapeamentos Transfinitos

O método de mapeamento transfinito está detalhadamente descrito em Fonseca (1989). A seguir, apresenta-se resumidamente os conceitos necessários à utilização do método em domínios bidimensionais.

Para descrever este mapeamento é necessário introduzir o conceito de projetor. Um projetor (P) é qualquer operador linear idempotente que mapeia uma superfície real (F) em uma superfície aproximada ($P[F]$), sujeito a certas restrições de interpolação. O projetor "lofting" é um dos projetores mais simples, ele efetua uma interpolação linear entre duas curvas do contorno, $\overrightarrow{\psi_1}(u)$ e $\overrightarrow{\psi_2}(u)$, de uma região F, dada por:

$$P[F] \equiv P(u,v) = (1-v)\overrightarrow{\psi_1}(u) + (v)\overrightarrow{\psi_2}(u) \tag{2.1}$$

para $0 \leqslant u \leqslant 1$ e $0 \leqslant v \leqslant 1$, onde u é uma coordenada paramétrica normalizada ao longo de $\overrightarrow{\psi_1}$ e $\overrightarrow{\psi_2}$, e v é uma coordenada normalizada que vale zero ao longo de $\overrightarrow{\psi_1}$ e um ao longo de $\overrightarrow{\psi_2}$ como mostra a figura 2.8.

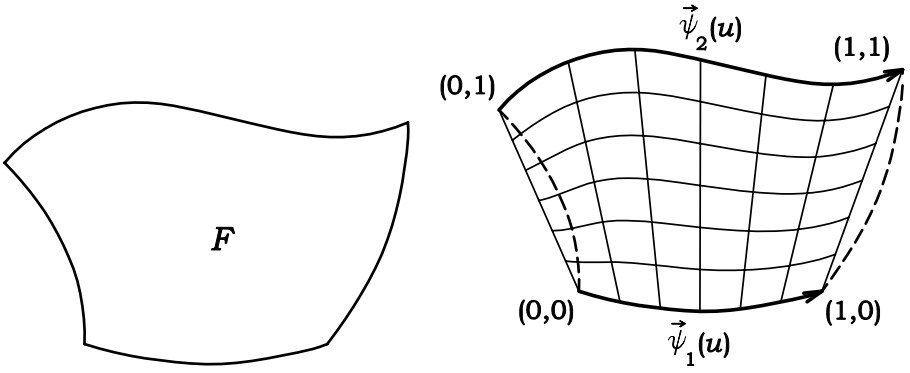
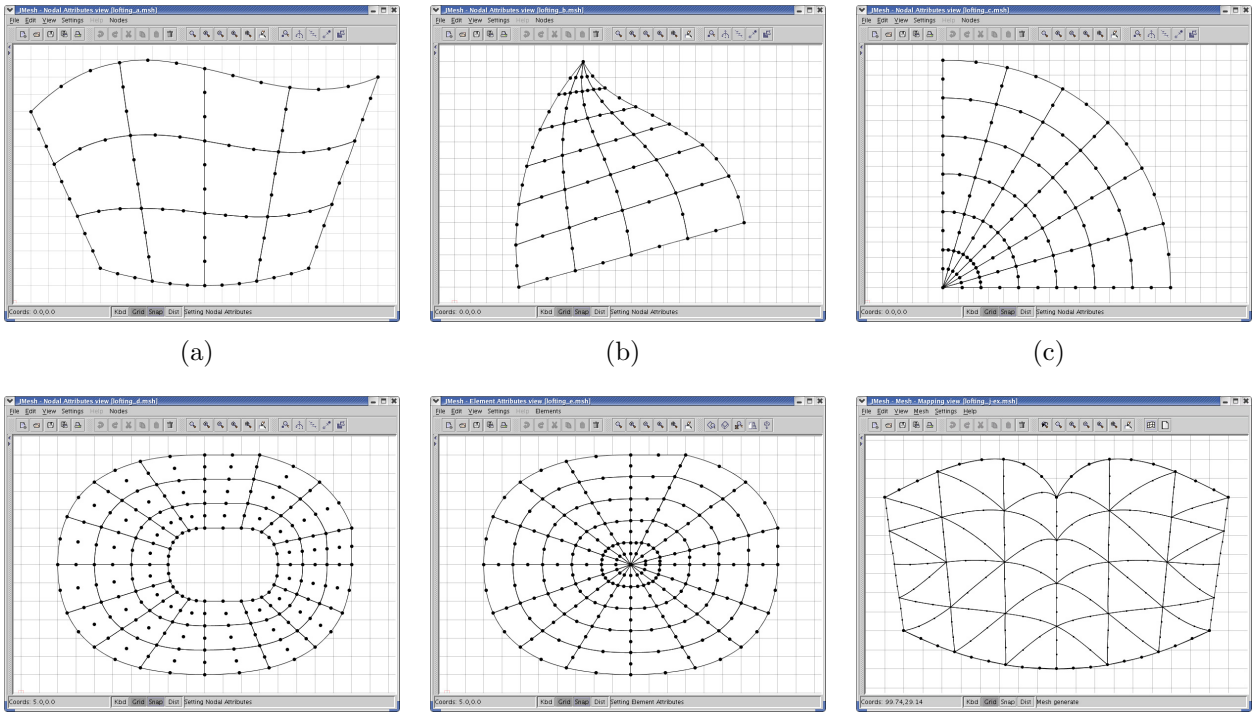


Figura 2.8: Representação do projetor "lofting"

Deve-se notar que, ao longo de $\overrightarrow{\psi_1}$ e $\overrightarrow{\psi_2}$, o contorno de $P[F]$ coincide com F de forma exata, e é uma aproximação linear nos demais lados. Salienta-se ainda que P é idempotente, isto é, $P(P[F]) = P[F]$.

Apesar de sua simplicidade, o mapeamento “lofting” é uma ferramenta poderosa na geração de malhas. Quando duas curvas abertas são usadas como contorno, uma região quadrilateral é criada (figura 2.9a). Quando duas curvas abertas com um ponto extremo comum são usadas, uma região triangular com dois lados curvos e um lado reto é gerada (figura 2.9b). Se uma das curvas degenera em um único ponto, uma região triangular com dois lados retos e um curvo é definida (figura 2.9c). Duas curvas fechadas e concêntricas definem uma área com um furo (figura 2.9d). Se uma curva fechada e uma curva degenerada em um ponto são usadas, uma malha radial é criada (figura 2.9e). Fortes descontinuidades de tangentes no contorno não afetam adversamente a qualidade da malha gerada (figura 2.9f). Deve-se sempre avaliar a consistência do que está sendo gerado para evitar que surjam nós redundantes em certos casos.



(a) (b) (c)

(d) (e) (f)

Figura 2.9: Exemplos de malhas geradas por mapeamentos “lofting”

Utilizando-se dois projetores “lofting”, um interpolando na direção u e outro na direção v, pode-se fazer combinações para obter projetores mais complexos que irão coincidir com o contorno da região F em todos os pontos.

Sejam os projetores "lofting":

$$P_1[F] \equiv P_1(u,v) = (1-v)\overrightarrow{\psi_1}(u) + (v)\overrightarrow{\psi_2}(u) \tag{2.2}$$
$$P_2[F] \equiv P_2(u,v) = (1-u)\overrightarrow{\xi_1}(v) + (u)\overrightarrow{\xi_2}(v)$$

Os projetores P_1 e P_2 são mostrados nas figura 2.10b e 2.10c, respectivamente. O projetor produto $P_1P_2[F] = P_2P_1[F]$ é mostrado na figura 2.10d, este projetor coincide exatamente com F nos quatro vértices, com aproximações lineares nos lados. Finalmente define-se o projetor soma booleana $(P_1 \oplus P_2)[F]$ de forma que F é mapeada de maneira precisa nos contornos (figura 2.10e):

$$(P1 \oplus P2)[F] \equiv P_1[F] + P_2[F] - P_1P_2[F] = (1-v)\overrightarrow{\psi_1}(u) + \tag{2.3}$$
$$+ v\overrightarrow{\psi_2}(u) + (1-u)\overrightarrow{\xi_1}(v) + u\overrightarrow{\xi_2}(v) - (1-u)(1-v)F(0,0) +$$
$$- (1-u)vF(0,1) - uvF(1,1) - u(1-v)F(1,0)$$

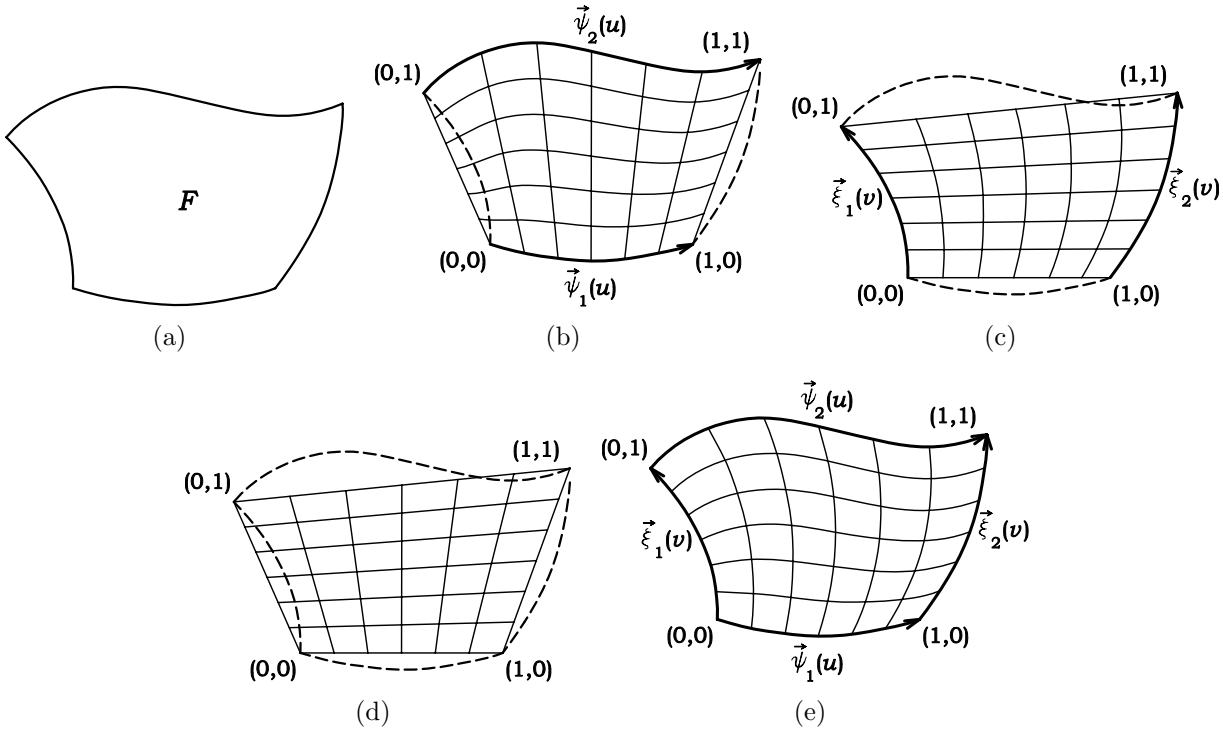
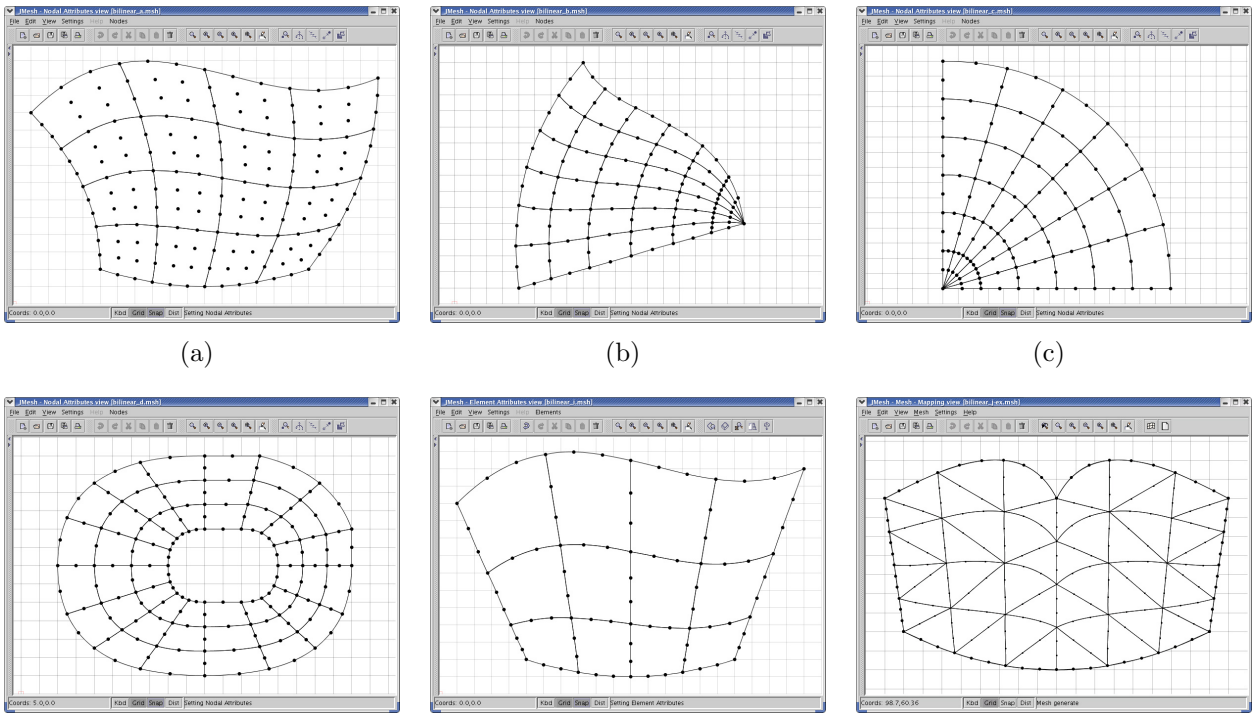


Figura 2.10: Representação de projetores em regiões quadrilaterais

O projetor $(P_1 \oplus P_2)[F]$ representa um sistema de coordenadas curvilíneas criado pelo mapeamento de um quadrado unitário em F. Este projetor pode ser chamado de interpolador Lagrangeano bilinear transfinito de F.

O mapeamento transfinito bilinear é mais adequado em regiões com contornos curvos nos quatro lados (figura 2.11a). As linhas interiores da malha possibilitam uma transição suave entre as curvas do contorno. Um lado curvo pode degenerar em um ponto para criar regiões com três lados (figura 2.11b e 2.11c). Na figura 2.11e pode-se ver como um espaçamento adequado dos pontos das curvas do contorno permite o refinamento local de uma região da malha. Fortes descontinuidades de tangentes no contorno não afetam adversamente a qualidade da malha gerada (figura 2.11f). Elementos quadrilaterais podem ser subdivididos em suas diagonais para formar elementos triângulares (figura 2.11f).



(a) (b) (c)

(d) (e) (f)

Figura 2.11: Exemplos de malhas geradas por mapeamentos transfinitos bilineares

Através do uso de interpoladores trilineares em um sistema de coordenadas triangulares, obtém-se um projetor capaz de mapear um triângulo unitário em uma região definida por três curvas no contorno.

Para uma região triangular T delimitada por três curvas $\overrightarrow{\psi}(u)$, $\overrightarrow{\xi}(v)$ e $\overrightarrow{\eta}(w)$, sendo u, v e w o sistema de coordenadas de área triangular normalizada, estabelecido no interior da região, define-se o seguinte conjunto de projetores lineares (figura 2.12):

$$N_1 \equiv N_1(u,v,w) = \left(\frac{u}{1-v}\right)\overrightarrow{\xi}(v) + \left(\frac{w}{1-v}\right)\overrightarrow{\eta}(1-v) \tag{2.4}$$
$$N_2 \equiv N_2(u,v,w) = \left(\frac{v}{1-w}\right)\overrightarrow{\eta}(w) + \left(\frac{u}{1-w}\right)\overrightarrow{\psi}(1-w)$$
$$N_3 \equiv N_3(u,v,w) = \left(\frac{w}{1-u}\right)\overrightarrow{\psi}(u) + \left(\frac{v}{1-u}\right)\overrightarrow{\xi}(1-u)$$

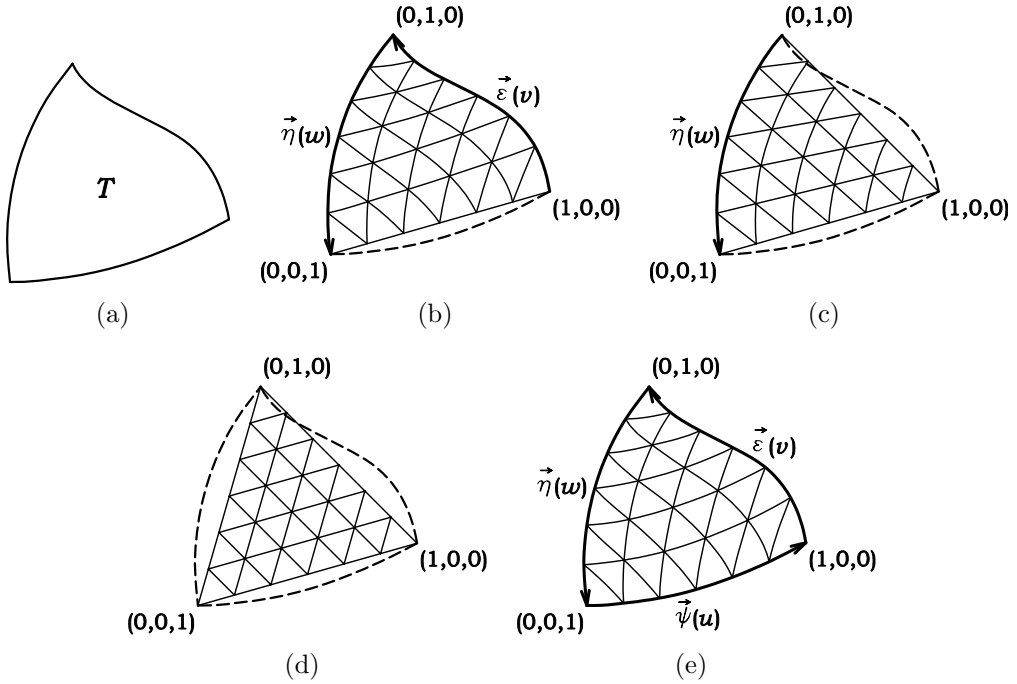


Figura 2.12: Representação de projetores em regiões triangulares

Estes projetores são interpolações Lagrangeanas lineares entre três pares de curvas de contorno. O projetor N_1 é ilustrado na figura 2.12b. O projetor N_1N_2 é ilustrado na figura 2.12c.

Deve-se observar que $L \equiv N_iN_jN_k$, $i \neq j \neq k$, projeta em um plano passando pelos três vértices de T (figura 2.12d). Pode-se obter ainda seis projetores soma quase-Booleana:

$$N_i \oplus N_j \equiv N_i + N_j - N_iN_j \tag{2.5}$$

com $i \neq j$; $i,j = 1,2,3$.

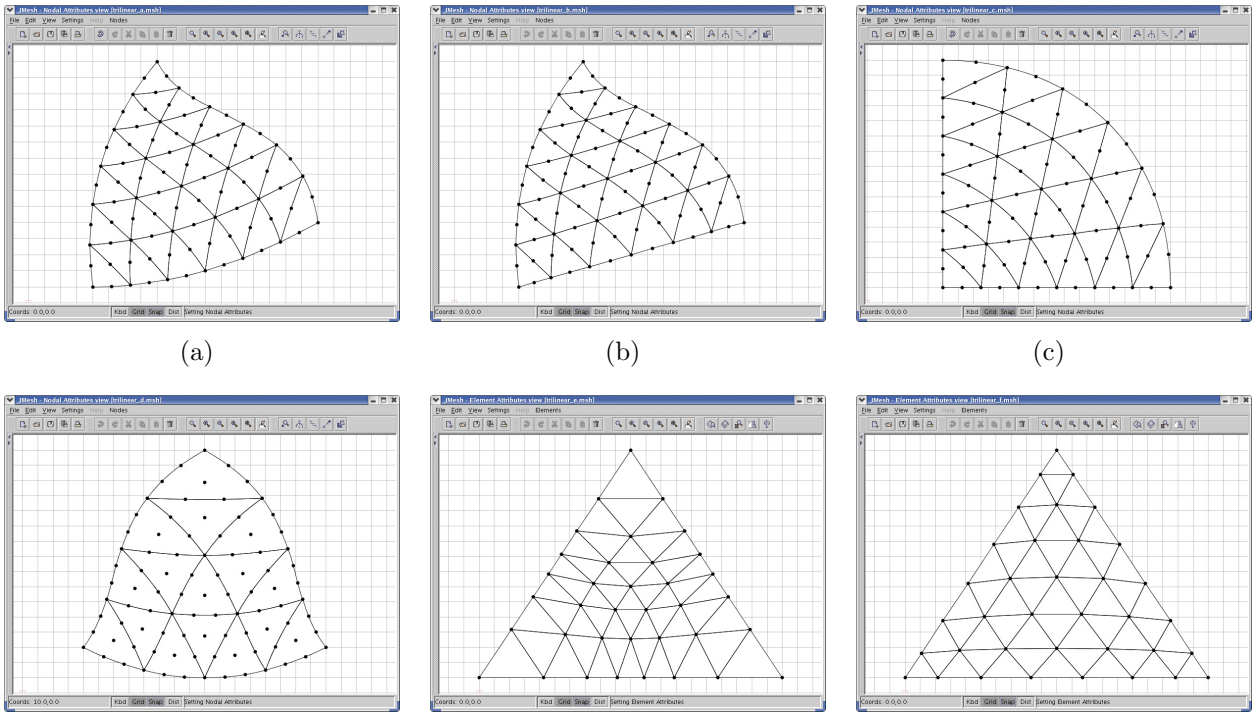
Cada um interpola T de forma exata no contorno. Um projetor trilinear pode ser então definido (figura 2.12e):

$$S \equiv \frac{1}{2}\left[(N_i \oplus N_j) + (N_i \oplus N_k)\right] = \frac{1}{2}\left[N_1 + N_2 + N_3 - L\right] = \tag{2.6}$$

$$S(u,v,w) = \frac{1}{2}\left[\left(\frac{u}{1-v}\right)\xi(v) + \left(\frac{v}{1-u}\right)\xi(1-u) + \left(\frac{v}{1-w}\right)\eta(w) + \left(\frac{w}{1-v}\right)\eta(1-v) + \left(\frac{w}{1-u}\right)\psi(u) + \left(\frac{u}{1-w}\right)\psi(1-w) - u\,\xi(0) - v\,\eta(0) - w\,\psi(0)\right]$$

com $i \neq j \neq k \neq i$.

Os projetores "lofting" e bilinear criam uma partição natural da região, composta de elementos quadrilaterais, enquanto o projetor trilinear cria elementos triangulares (figura 2.13).



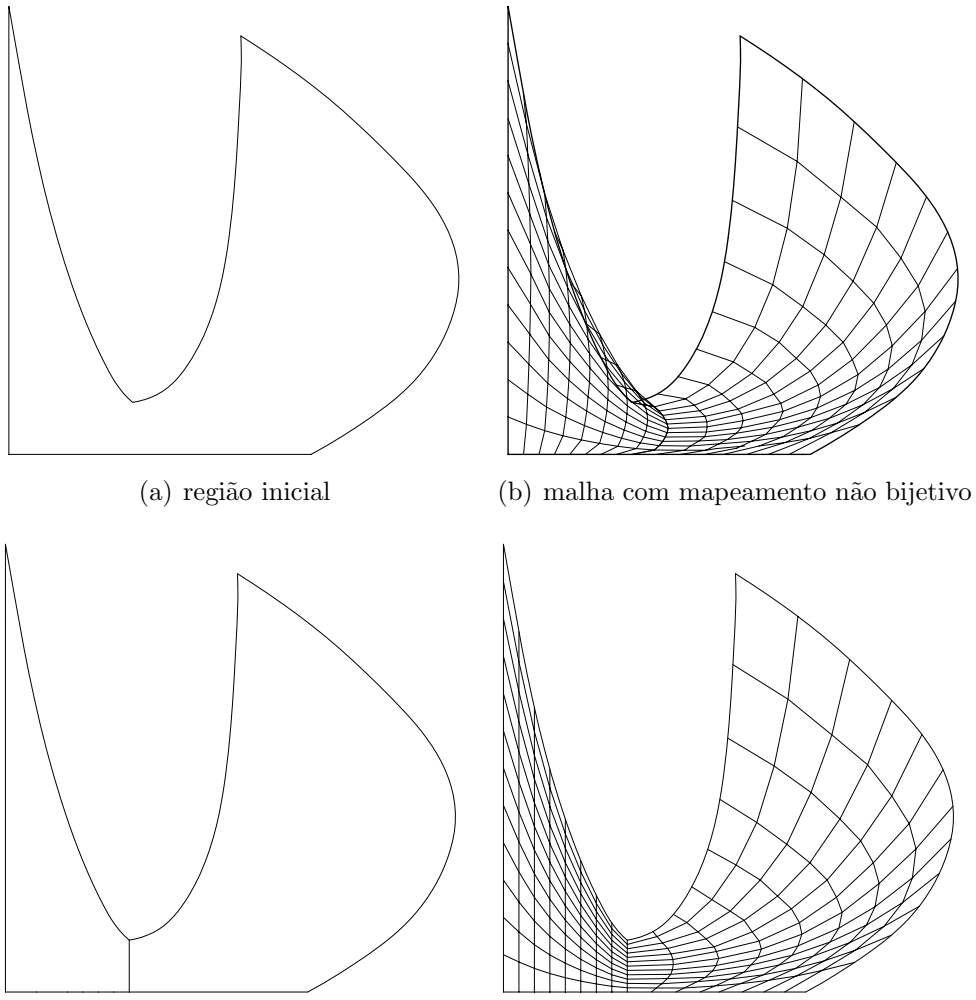
(a) (b) (c)

(d) (e) (f)

Figura 2.13: Exemplos de malhas geradas por mapeamentos transfinitos trilineares

Quando os mapeamentos "lofting" e bilinear são usados para mapear regiões triangulares, a malha resultante fica adensada em torno do lado que se degenera em um ponto (figuras 2.9b, 2.9c, 2.11b e 2.11c). Esse adensamento pode ser evitado usando o mapeamento transfinito trilinear para gerar malhas em regiões limitadas por três curvas (figura 2.13).

Quando uma curva muito distorcida é usada para definir o contorno de uma região (figura 2.14a), o problema de mapeamento não bijetivo pode aparecer. Quando este problema ocorre, as linhas da malha de elementos finitos cruzam para fora da região delimitada pelo contorno (figura 2.14b).



(a) região inicial (b) malha com mapeamento não bijetivo

(c) região subdividida em 2 sub-regiões (d) malha com mapeamento bijetivo

Figura 2.14: Exemplo de malha com mapeamento não bijetivo

Uma solução para esse problema é dividir as regiões muito distorcidas em duas ou mais regiões com geometria mais regular (figura 2.14c).

Existem duas alternativas para representação do contorno de regiões: a forma contínua e a forma discreta. A forma contínua representa o vetor posição de uma curva do contorno como função de alguma coordenada paramétrica. Esta representação permite que uma posição arbitrária da curva possa ser obtida em qualquer ponto da mesma.

A representação discreta consiste em listas finitas de pontos localizados na curva, com uma única coordenada associada a cada ponto da lista. Uma posição na curva pode somente ser avaliada nos pontos contidos na lista e é indefinida nos demais pontos. A forma discreta de representação é inteiramente geral e pode ser usada para qualquer curva.

Curvas complexas podem ser criadas pela concatenação de uma série de curvas simples sem acréscimo de complexidade nas rotinas de mapeamento. Para evitar que uma grande quantidade de dados tenha que ser informada para descrever uma curva, pode-se usar um gerador de curvas que cria automaticamente descrições discretas de curvas baseadas em vários modelos contínuos.

Inúmeras vantagens tornam o mapeamento transfinito muito adequado para servir a um sistema interativo de geração de malhas de elementos finitos:

1. Pontos no interior da malha são gerados por uma expressão fechada, extremamente simples;

2. A quantidade de cálculos requerida para uma malha de pontos não aumenta com a complexidade das curvas do contorno, caracterizando a eficiência computacional;

3. A curvatura das curvas do contorno e o espaçamento dos nós no contorno são refletidos de forma precisa e previsível na malha gerada;

4. As curvas do contorno não necessitam ser funções simples;

5. Ângulos agudos e pontos de inflexão não representam nenhum problema específico;

6. Nenhum erro no ajuste de curvas é introduzido além dos erros de discretização inerentes ao método dos elementos finitos;

7. O uso de geradores automáticos de curvas de contorno reduz os dados de entrada;

8. Em geral, menos regiões e curvas de contorno são requeridas para mapeamentos transfinitos do que para outros métodos;

9. Uma única rotina para cada mapeamento manipula todo tipo de curva de contorno.

A maior desvantagem deste método é a restrição imposta na topologia da malha:

1. Nas regiões “lofting” e bilineares o número de divisões deve ser o mesmo em curvas opostas;

2. Nas regiões trilineares o número de divisões deve ser o mesmo em todas as curvas.

Regiões especiais de transição são necessárias em alguns casos para superar esta dificuldade.

Capítulo 3

RECURSOS UTILIZADOS NO DESENVOLVIMENTO DA APLICAÇÃO

3.1 Paradigma de Programação Orientada a Objetos

3.1.1 Coleções de Objetos

Muitas aplicações para as quais desejam-se desenvolver um programa, consistem em sistemas bastante complexos. Uma maneira natural de lidar com a complexidade de um sistema é dividi-lo em subsistemas mais simples, de maneira que o comportamento do sistema, como um todo, possa ser expresso em termos dos comportamentos de seus subsistemas e das interações entre eles.

Para que essa abordagem possa ser espelhada diretamente em um programa para simulação ou controle de um sistema, linguagens de programação mais modernas oferecem recursos para construção de um programa como uma coleção de componentes de programa, com interfaces bem definidas, que especificam as interações entre esses componentes.

No paradigma de programação orientado por objetos, a construção de um programa para implementação de um determinado sistema baseia-se em uma correspondência natural e intuitiva entre esse sistema e a simulação do comportamento do mesmo: a cada entidade do sistema corresponde, durante a execução do programa, *um objeto*, com atributos e comportamento descritos por um componente desse programa.

O desenvolvimento de *software* para implementação de um sistema envolve fases de análise, projeto e implementação desse sistema. O princípio em que se baseia o paradigma de orientação por objetos, o de que existe uma correspondência entre componentes do sistema e objetos, torna mais simples esse processo. Objetos constituem limites naturais para construções de *abstrações de dados*: todas as informações referentes a uma dada entidade são confinadas em um determinado objeto, que se relaciona com outros objetos mediante uma interface bem definida.

A maioria das linguagens de programação orientadas por objetos usa o conceito de *classe*, para descrição de grupos de objetos semelhantes. Um programa nessas linguagens consiste em uma coleção de definições de classes, que descrevem os objetos que implementam entidades de um sistema (Camarão & Figueiredo 2003).

3.1.2 Classes e Objetos

Uma classe é um componente de programa que descreve a "estrutura" e o "comportamento" de um grupo de objetos semelhantes - isto é, as informações que caracterizam o estado desses objetos e as ações (ou operações) que eles podem realizar. Os objetos de uma classe - também chamados de *instâncias* da classe - são criados durante a execução de programas.

Uma classe é formada, essencialmente, por *construtores de objetos* dessa classe, variáveis e métodos. A criação de um objeto dessa classe consiste na criação de cada uma das variáveis do objeto, especificadas na classe. Os valores armazenados nessas variáveis determinam o *estado* do objeto. Uma variável de um objeto é também chamada de "atributo" desse objeto.

Objetos podem "receber mensagens", sendo uma mensagem basicamente uma chamada a um método específico de um objeto, que realiza uma determinada operação, em geral dependente do estado desse objeto. A execução de uma chamada a um método de um objeto pode modificar o estado desse objeto, isto é, modificar os valores dos seus atributos, e pode retornar um resultado (Camarão & Figueiredo 2003).

3.1.3 Abstração

Abstrair significa decompor um sistema complicado em suas partes fundamentais e descrevê-las em uma linguagem simples e precisa. A descrição das partes de um sistema implica atribuir-lhes um nome e descrever suas funcionalidades. Por exemplo, a interface gráfica com o usuário de um editor de textos compreende a abstração de um menu "editar" que oferece várias opções de edição de texto incluindo recortar e colar porções de texto ou outros objetos gráficos. Sem entrar em detalhes sobre como uma interface gráfica com o usuário representa e exibe textos ou objetos gráficos, os conceitos de "recortar" e "colar" são simples e precisos. Uma operação de recorte apaga o texto ou gráfico selecionado e o coloca em uma área de armazenamento externa. A operação de colagem insere o conteúdo externamente armazenado em uma localização específica do texto. Dessa forma, a funcionalidade abstrata do menu "editar" e suas operações de recortar e colar são definidas em uma linguagem precisa o suficiente para ser clara e simples o bastante para "abstrair" os detalhes desnecessários. Essa combinação de clareza e simplicidade traz benefícios a robustez, uma vez que leva a implementações corretas e compreensíveis (Goodrich & Tamassia 2002).

3.1.4 Encapsulamento

Outro princípio importante em projeto orientado a objetos é o conceito de *encapsulamento*. Este princípio estabelece que os diferentes componentes de um sistema de *software* não devem revelar detalhes internos de suas respectivas implementações. Genericamente, o princípio do encapsulamento propõe que todos os componentes de um grande sistema de *software* operem dentro de uma filosofia de conhecer o mínimo necessário sobre os demais.

Uma das maiores vantagens do encapsulamento é que ele oferece ao programador liberdade na implementação dos detalhes do sistema. A única restrição ao programador é manter a interface abstrata que é percebida pelos de fora. Por exemplo, o programador do código do menu "editar" da interface gráfica com o usuário de um editor de textos pode, em um primeiro momento, implementar as operações de copiar e colar copiando e restaurando telas para a área externa de armazenamento. Mais tarde, pode ficar insatisfeito com essa implementação, uma

vez que não permite um armazenamento compacto da seleção e não distingue objetos gráficos de textos. Se o programador tiver projetado a interface das operações de copiar e colar tendo em mente o encapsulamento, trocar a implementação por uma que armazene o texto como texto e os objetos gráficos em uma forma compacta apropriada não irá causar nenhum problema aos métodos que necessitam interagir com esta interface gráfica. Dessa forma, encapsulamento permite a adaptação, porque autoriza a alteração de detalhes de partes de um programa sem afetar de forma negativa outros componentes (Goodrich & Tamassia 2002).

3.1.5 Modularidade

Além de abstração e encapsulamento, outro princípio fundamental de projeto orientado a objetos é a *modularidade*. Sistemas modernos de *software* normalmente estão compostos por vários componentes diferentes que devem interagir corretamente, fazendo com que o sistema como um todo funcione de forma adequada. Para manter essas interações corretas é necessário que os diversos componentes estejam bem organizados. Na abordagem orientada a objetos, essa organização se centra no conceito de *modularidade*. A modularidade refere-se a uma estrutura de organização na qual os diferentes componentes de um sistema de *software* são divididos em unidades funcionais separadas.

A estrutura imposta pela modularidade auxilia a tornar o *software* reutilizável. Se os módulos do *software* forem escritos de uma forma abstrata para resolver problemas genéricos, então os módulos podem ser reutilizados quando instâncias do mesmo problema geral surgirem em outros contextos (Goodrich & Tamassia 2002).

3.1.6 Herança

Para evitar código redundante, o paradigma de orientação a objetos oferece uma estrutura hierárquica e modular para reutilização de código através de uma técnica conhecida como *herança*. Esta técnica permite projetar classes genéricas que podem ser especializadas em classes mais particulares, onde as classes especializadas reutilizam o código das mais genéricas. A classe genérica, também conhecida por *classe base* ou *superclasse*, define variáveis de instância

"genéricas" e métodos que se aplicam em uma variada gama de situações. A classe que *especializa*, ou *estende* ou *herda de* uma superclasse não necessita fornecer uma nova implementação para os métodos genéricos, uma vez que os herda. Deve apenas definir aqueles métodos que são especializados para esta *subclasse* em particular (também conhecida como classe *derivada*) (Goodrich & Tamassia 2002).

3.1.7 Polimorfismo

Como bem destacado por Goodrich e Tamassia(2002) "polimorfismo" significa, literalmente, "muitas formas". No contexto de projeto orientado a objetos, entretanto, refere-se à habilidade de uma variável de objeto de assumir formas diferentes. Linguagens orientadas a objetos, referenciam objetos usando variáveis referência. Uma variável referência o deve especificar que tipo de objeto ela é capaz de referenciar em termos de uma classe S. Isso implica, entretanto, que o também pode se referir a qualquer objeto pertencente à classe T, derivada de S. Analise agora o que acontece se S define um método a() e T também define um método a(). A seqüência de ativação de métodos sempre é iniciada com a busca pela classe mais restritiva à qual se aplica. Ou seja, quando o se refere a um objeto da classe T e o.a() é invocado, então será ativada a versão de T do método a(), em lugar da versão de S. Neste caso, diz-se que T ***sobrescreve*** o método a() de S. Por outro lado, se o se refere a um objeto da classe S(que, ao contrário, não é um objeto da classe T), quando o.a() for ativado, será executada a versão de S de a(). Um polimorfismo como esse é útil porque aquele que chama o.a() não precisa saber quando o se refere a uma instância de T ou S para poder executar a versão correta de a(). Dessa forma, a variável de objeto o pode ser *polimórfica*, ou assumir muitas formas, dependendo da classe específica dos objetos aos quais está se referindo. Esse tipo de funcionalidade permite a uma classe especializada T estender uma classe S, herdar os métodos genéricos de S e redefinir outros métodos de S, de maneira que sejam incluídos como propriedades específicas dos objetos T.

Algumas linguagens orientadas a objetos também oferecem um tipo de polimorfismo "em cascata", que é mais precisamente conhecido como ***sobrecarga*** de métodos. A sobrecarga

ocorre quando uma única classe T tem vários métodos com o mesmo nome, desde que cada um tenha uma *assinatura* diferente. A assinatura de um método é uma combinação entre seu nome e o tipo e a quantidade de argumentos que são passados para o mesmo. Dessa forma, mesmo que vários métodos de uma classe tenham o mesmo nome, eles são distinguíveis pelo compilador pelo fato de terem diferentes assinaturas, ou seja, na verdade são desiguais. Em linguagens que possibilitam a sobrecarga de métodos, o ambiente de execução determina qual método ativar para uma determinada chamada de método que percorre a hierarquia de classes em busca do primeiro método cuja assinatura combine com a do método que está sendo invocado. Por exemplo, imagine uma classe T que define o método a(), derivada da classe U que define o método a(x,y). Se um objeto o da classe T recebe a mensagem "o.a(x,y)", então a versão de U do método a() é ativada (com os dois parâmetros x e y). Assim, o verdadeiro polimorfismo aplica-se apenas a métodos que têm a mesma assinatura mas estão definidos em classes diferentes.

A herança, o polimorfismo e a sobrecarga de métodos suportam o desenvolvimento de *software* reutilizável. Pode-se estabelecer classes que herdam as variáveis e os métodos de instância genéricos e que podem, a seguir, definir novas variáveis e métodos de instância mais específicos que lidam com os aspectos particulares dos objetos da nova classe (Goodrich & Tamassia 2002).

3.2 Padrões de Projeto de *Software*

Padrões de projeto podem ser definidos como a estrutura básica de um projeto de *software* bem-sucedido, capaz de fornecer um esquema para os subsistemas ou componentes de um sistema de *software* a ser projetado. Esse esquema deve ser específico para resolver o problema em questão e suficientemente genérico para atender a futuros problemas e requisitos. A utilização de Padrões de Projeto propicia evitar ou minimizar o re-projeto.

A utilização de Padrões de Projeto (Gamma, Helm, Johnson & Vlissides 1995) em sistemas orientados a objetos torna estes sistemas mais flexíveis e reutilizáveis. Os Padrões de Projeto ajudam os projetistas de *software* a reutilizar bons projetos ao basear os novos projetos em

experiências anteriores. O uso de determinado Padrão de Projeto garante que uma grande quantidade de decisões de projeto decorra automaticamente, porque o sistema em desenvolvimento será baseado em um sistema que está em funcionamento.

Entre as vantagens de se utilizar padrões de projeto no desenvolvimento de *software* pode-se citar: aumento de produtividade, uniformidade na estrutura do *software*, incremento da padronização no desenvolvimento de *software*, aplicação imediata por outros desenvolvedores, redução da complexidade do sistema (Pietro 2001).

3.2.1 Padrão *Model-View-Controller*

Visando separar o modelo de sua representação, a implementação do pré-processador do `INSANE` é baseada no padrão de projeto *Model-View-Controller (MVC)*. A utilização desta metáfora de programação permite que o controle da geração da malha, através de interação com o usuário, e a visualização da mesma sejam implementados independentemente do modelo adotado, minimizando as tarefas de manutenção e expansão da aplicação. A implementação segundo o padrão *MVC* permite o aperfeiçoamento gradual da aplicação através de mudança de plataforma, criação de diversas vistas sincronizadas com o modelo, substituição ou atualização das diversas vistas e disponibilização "on-line" do sistema.

Existe um ciclo de vida para cada uma das atividades executadas pelo programa. Este ciclo permite que o usuário faça alterações no modelo e visualize o resultado a cada alteração, até que consiga o resultado desejado. O referido ciclo compõe-se de: especificação do usuário, atualização do modelo e visualização.

O padrão de projeto *Model-View-Controller* pode ser usado para a implementação do ciclo de vida de cada atividade. Este padrão divide a aplicação em três componentes: `modelo`, `vista` e `controlador`. A Figura 3.1 ilustra o padrão.

O `modelo` contém o núcleo dos dados e da funcionalidade do sistema, sendo independente das saídas e entradas de dados. A `vista` apresenta para o usuário as informações armazenadas no `modelo`. Cada `controlador` é associado a um componente `vista`, sendo o responsável pela percepção das entradas do usuário e tradução das mesmas em requisições de serviços para os

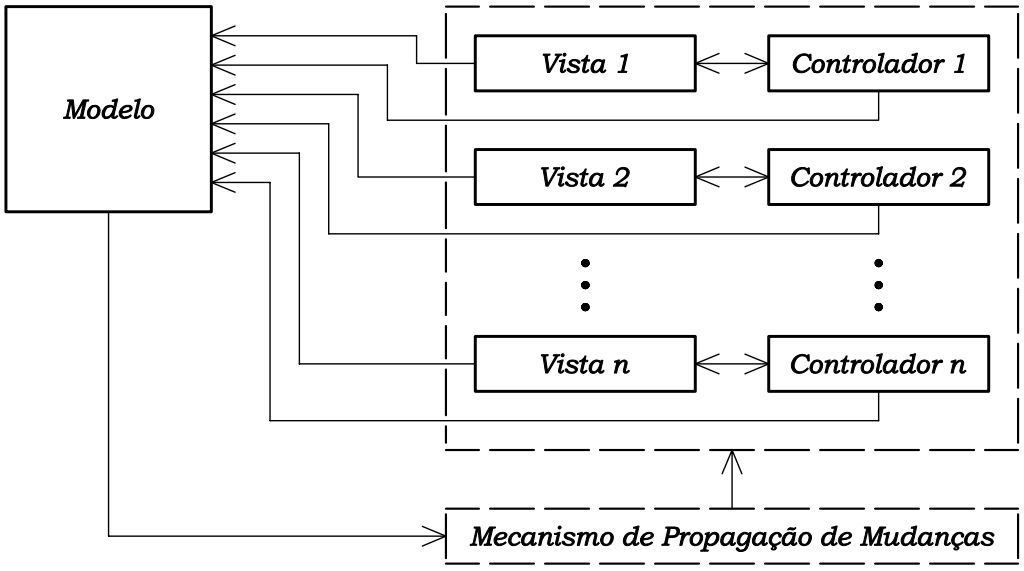


Figura 3.1: Componentes do padrão `MVC`

componentes `modelo` e `vista`. Todas as requisições dos usuários devem ser feitas através dos controladores (Buschmann, Meunier, Rohnert, Sommerlad & Stal 1995).

Existe um mecanismo de propagação de mudanças que garante a consistência e a comunicação entre os componentes `controlador` e `vista` com o componente `modelo`. No momento em que um componente `controlador` ou `vista` é criado, o mecanismo de propagação de mudanças efetua seu registro, ligando-o ao `modelo` do qual ele é dependente. Os componentes `vista` e `controlador` dependem desse registro para serem informados das atualizações do `modelo`.

O mecanismo de propagação de mudanças é disparado a cada mudança de estado do `modelo`, acarretando na execução do procedimento de atualização do componente `vista`, que exibe ao usuário a informação atualizada. Cada `vista` é associada a um único `controlador` e fornece a este a funcionalidade necessária para manipular a exibição de dados.

3.2.2 Padrão `Command`

O uso do padrão `Command` na implementação do pré-processador do INSANE permite o encapsulamento de rotinas de execução em objetos, a associação destes objetos a elementos de interface gráfica com o usuário (GUI) e dispositivos de entrada (teclado e mouse), a execução

de uma mesma rotina disparada por diferentes elementos de GUI e possibilita um incremento na modularidade de seu código. O encapsulamento das rotinas de execução possibilita também que a realização de alterações nas mesmas não provoque modificações nas classes existentes.

O padrão `Command` baseia-se em uma classe abstrata de mesmo nome, a qual declara uma interface para execução de operações. Na sua forma mais simples, esta interface inclui uma operação abstrata `execute()`. As subclasses concretas de `Command` especificam um par receptor-ação através do armazenamento do receptor como uma variável de instância e pela implementação de `execute()`, para invocar a solicitação. O receptor tem o conhecimento necessário para poder executar a solicitação.

Este padrão desacopla o objeto que invoca a operação daquele que tem o conhecimento para executá-la. Isto proporciona grande flexibilidade no projeto da interface de usuário. Uma aplicação pode oferecer tanto uma interface gráfica com *menus* como uma interface gráfica com *botões* para algum recurso seu, simplesmente fazendo com que o *menu* e o *botão* compartilhem uma instância da mesma classe que implementa `Command`. O padrão `Command` possibilita a substituição dinâmica de comandos, o que é muito útil para interfaces gráficas sensíveis ao contexto. Pode-se ainda concatenar comandos para compor comandos maiores, propiciando a redução da complexidade destes. Todos este recursos são possíveis porque o objeto que emite a solicitação não precisa conhecer a execução da solicitação.

O tempo de vida de um objeto `Command` é independente de sua solicitação original, permitindo armazenar comandos e executar suas rotinas em momentos distintos. A operação `execute()`, de `Command`, pode armazenar estados para que o comando possa reverter seus efeitos. Para suportar a operação desfazer a interface de `Command` deve acrescentar a operação `undo()`, que reverte os efeitos de uma chamada anterior de `execute()`. Os comandos executados devem ser armazenados em uma lista histórica. O nível ilimitado de desfazer e refazer operações é obtido percorrendo esta lista para trás e para frente, chamando operações `undo()` e `execute()`, respectivamente (Gamma et al. 1995).

A figura 3.2 apresenta os componentes envolvidos com o padrão `Command` e ilustra o relacionamento entre eles. `Receiver` é o componente que sabe como executar as operações

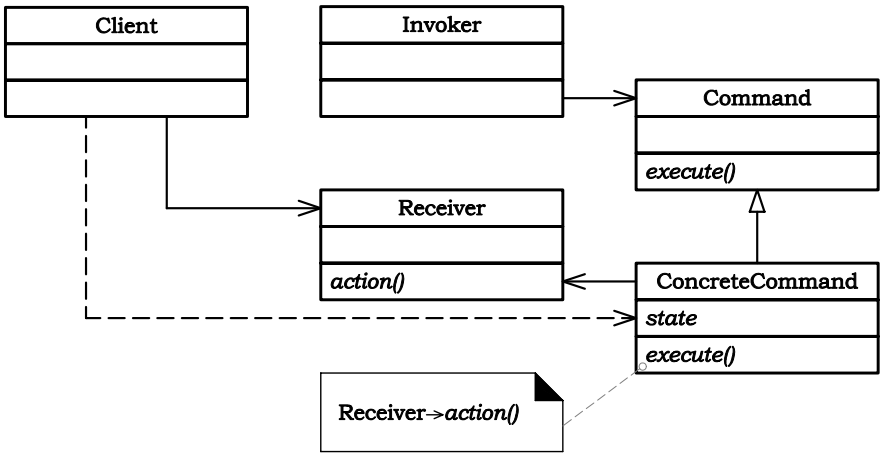


Figura 3.2: Estrutura do padrão *Command* (Gamma et al. 1995)

associadas a uma solicitação, qualquer classe pode funcionar como um `Receiver`. A classe `Command` declara uma interface para a execução de uma operação. `ConcreteCommand` implementa o processo `execute()` através da invocação das operações correspondentes no `Receiver` e define uma vinculação entre um objeto `Receiver` e uma ação. Quando os comandos podem ser desfeitos, `ConcreteCommand` armazena estados para desfazer o comando antes de invocar `execute()`. O componente `Invoker` é responsável por disparar o processo `execute` de seu comando associado (Gamma et al. 1995). O componente `Client` cria um objeto `ConcreteCommand`, o associa a um `Invoker` e estabelece o seu receptor (`Receiver`).

3.3 Linguagem Java

Dentre as linguagens que suportam o paradigma de programação orientada a objetos, uma das mais utilizadas é Java. Alguns aspectos de Java são particularmente relevantes e foram analisados durante o processo de escolha da linguagem para implementação deste trabalho de dissertação. Estes aspectos são:

1. Independência de sistema operacional;
2. Performance numérica;
3. Capacidade de reutilização do *software*;
4. Suporte à persistência dos dados;

3.3.1 Portabilidade

Java independe do sistema operacional, pois utiliza um processo diferente da compilação ou interpretação tradicionalmente conhecidos.

Um interpretador é, como o nome indica, um programa que interpreta diretamente as frases do programa fonte, isto é, simula a execução dos comandos desse programa sobre um conjunto de dados, também fornecidos como entrada para o interpretador. A interpretação de programas escritos em uma determinada linguagem define uma "máquina virtual", na qual é realizada a execução de instruções dessa linguagem.

A interpretação de um programa em linguagem de alto nível pode ser centenas de vezes mais lenta do que a execução do código objeto gerado para esse programa pelo compilador. A razão disso é que o processo de interpretação envolve simultaneamente a análise e simulação da execução de cada instrução do programa, ao passo que essa análise é feita previamente, durante a compilação, no segundo caso. Apesar de ser menos eficiente, o uso de interpretadores muitas vezes é útil principalmente devido ao fato de que, em geral, é mais fácil desenvolver um interpretador do que um compilador para uma determinada linguagem.

Esse aspecto foi explorado pelos projetistas da linguagem Java, no desenvolvimento de sistemas (ou ambientes) para programação e execução de programas nessa linguagem: esses ambientes são baseados em uma combinação dos processos de compilação e interpretação. Um ambiente de programação Java é constituído de um compilador Java, que gera um código de mais baixo nível, chamado de *bytecodes*, que é então interpretado. Um interpretador de *bytecodes* interpreta instruções da chamada "Máquina Virtual Java", nome abreviado como *JVM*. Esse esquema usado no ambiente de programação Java não apenas contribuiu para facilitar a implementação da linguagem em um grande número de computadores diferentes, mas constitui uma característica essencial no desenvolvimento de aplicações voltadas para a Internet, pois possibilita que um programa compilado em determinado computador possa ser transferido através da rede e executado em qualquer outro computador que disponha de um interpretador de *bytecodes* (Camarão & Figueiredo 2003).

3.3.2 Capacidade de Reutilização de *Software* em Java

Programadores Java concentram-se na elaboração de novas classes e reutilização de classes existentes. Existem muitas bibliotecas de classe e outras estão sendo desenvolvidas em todo o mundo. O *software* é, então, construído a partir de componentes amplamente disponíveis, portáveis, bem-documentados, cuidadosamente testados e bem definidos. Esse tipo de capacidade de reutilização de *software* acelera o desenvolvimento de programas poderosos e de alta qualidade (Deitel & Deitel 2003).

Para perceber o potencial completo da capacidade de reutilização de *software*, precisamos aprimorar os esquemas de catalogação, os esquemas de licença, os mecanismos de proteção que asseguram que as cópias-mestras das classes não sejam corrompidas, os esquemas de descrição que projetistas de sistema utilizam para determinar se objetos existentes atendem às necessidades, os mecanismos de navegação que determinam as classes que estão disponíveis e o grau em que elas atendem aos requisitos de desenvolvimento de *software*, e assim por diante. Muitos problemas interessantes de pesquisas e desenvolvimento foram solucionados e muitos outros necessitam ser resolvidos. Esses problemas acabarão sendo resolvidos de uma forma ou de outra, uma vez que o valor potencial da reutilização de *software* em Java é enorme (Deitel & Deitel 2003).

3.4 Persistência de Dados com XML

O armazenamento de dados em variáveis e arranjos (vetores e matrizes) é temporário - os dados são perdidos quando uma variável local "sai do escopo" ou quando o programa termina. Arquivos são utilizados para retenção a longo prazo de grandes quantidades de dados, mesmo depois de terminar a execução do programa que criou os dados. Os dados mantidos em arquivos são freqüentemente chamados de dados persistentes (Deitel & Deitel 2003).

A adoção da web como veículo de acesso a sistemas de informação trouxe novamente a preocupação com a estrutura dos documentos. Primeiro, para fornecer o mesmo conteúdo em formatos alternativos, personalizados para computadores desktop, celulares, auto-atendimento

telefônico ou para impressão em papel; segundo, para possibilitar o acesso às informações por outras aplicações, em vez de apenas por usuários humanos (Lozano 2003).

O problema então era criar uma linguagem capaz de descrever qualquer tipo de documento, exigindo extensibilidade, mas preservando tanto a facilidade de autoria do HTML para seres humanos quanto a sua sintaxe simples para processamento por *software*. Esta linguagem é a XML(eXtensible Markup Language).

Estudando as formas disponíveis atualmente para armazenar dados persistentes, a mais indicada para implementação deste trabalho é a XML. Ela é um formato padronizado de arquivo texto, projetado para escrever e estruturar dados.

A XML não é apenas "mais uma forma" de gerar sites web, seu grande diferencial está na possibilidade de se processar a informação contida no documento original, ignorando a formatação fornecida pelas folhas de estilo, tornando-se um formato universal para importação e exportação de dados.

A XML gerou um conjunto de tecnologias rico e útil para uma vasta gama de aplicações. Não há revolução alguma na XML, mas apenas novas maneiras de realizar tarefas que já eram possíveis antes, com outras tecnologias. O diferencial é que as novas maneiras são portáveis, independentemente de linguagem de programação ou sistema operacional e baseadas em padrões abertos (Lozano 2003).

A plataforma de desenvolvimento Java oferece todas as API´s (Aplication Program Interfaces) necessárias à escrita de programas capazes de ler, criar e editar documentos XML. Tais API´s permitem a leitura e a escrita dos documentos em arquivos, conexões TCP/IP, Strings e outros meios de Entrada/Saída (Liesenfeld 2002).

Capítulo 4

ANÁLISE E PROJETO ORIENTADOS A OBJETOS

O processo de desenvolvimento de *software* orientado a objetos compreende três fases principais. Inicialmente, na fase de *análise*, procura-se enfatizar a descoberta e descrição dos objetos - ou conceitos - do domínio do problema. Em seguida, na fase de *projeto*, procura-se definir os elementos lógicos de *software*, seus atributos e métodos. Finalmente, durante a fase de *construção*, os componentes do projeto serão implementados em uma linguagem de programação que suporte o paradigma da progamação orientada a objetos. Na apresentação destas fases ao longo do texto, serão utilizados diversos diagramas que possibilitem a visualização da estrutura da aplicação. Os diagramas seguem a proposta da UML (Unified Modeling Language), linguagem padronizada para a modelagem de sistemas de *software* orientados a objetos.

UML é um meio para expressar projetos e refletir as melhores práticas da programação orientada a objetos. Essa linguagem de modelagem, projetada para ser independente do processo, tornou-se o modo-padrão para desenhar diagramas de projetos orientados a objetos, e também se espalhou em campos não orientados a objetos (Fowler & Scott 2000).

A essência da *análise* e do *projeto* orientados a objetos é enfatizar a consideração de um domínio de problema e uma solução lógica, segundo a perspectiva de objetos (coisas, conceitos ou entidades), conforme mostrado na figura 4.1. Nesta figura aparece o símbolo UML para *notas*. Estas representam comentários, observações e esclarecimentos que podem ser utilizadas para qualquer elementos da UML.

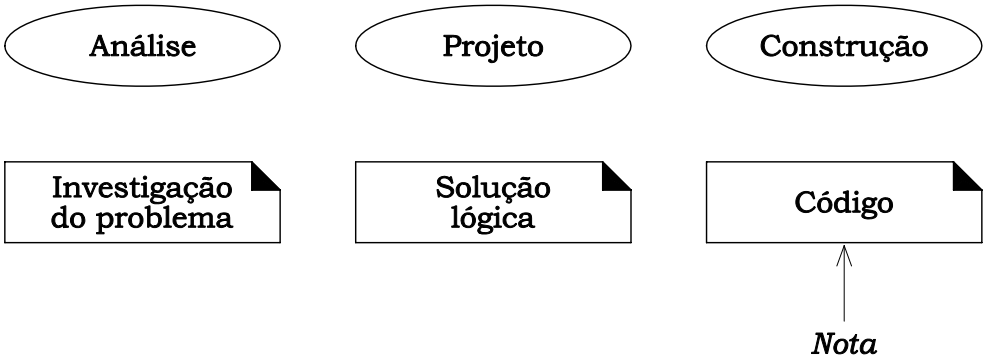


Figura 4.1: Fases de desenvolvimento

Este capítulo trata da *análise*, que enfatiza uma investigação do problema, de como uma solução é definida (Larman 2000). Para criar o *software* de uma aplicação, é necessária uma descrição do problema e dos seus requisitos - o que é o problema e o que o sistema deve fazer.

O projeto é a expansão e adaptação técnica dos resultados da análise. As classes, objetos, relacionamentos e colaborações da análise são complementados com novos elementos com o propósito de especificar como implementar o sistema em computador. Aos conceitos estabelecidos na análise, através do modelo conceitual, são acrescidas classes que ajudam estes conceitos a tornarem-se persistentes, a comunicarem-se e a apresentarem-se na interface com o usuário (Pagliosa 2004).

Este capítulo apresenta também o projeto orientado a objetos da aplicação. São discutidos os padrões de projeto utilizados, identificando as classes e instâncias participantes do projeto, bem como suas responsabilidades.

Para apresentar este projeto são utilizados diagramas de classes, que permite conhecer como as classes se comunicam para desempenhar suas funções, e diagramas de seqüência, que mostram várias atividades e como estas são desenvolvidas no decorrer do tempo.

4.1 Arquitetura em Camadas e Padrões de Projetos de *Software*

Visando separar o modelo de sua representação, a implementação do pré-processador do `INSANE` é baseada no padrão de projeto *Model-View-Controller* (seção 3.2.1 na página 26).

Suas principais vantagens consistem em facilitar a manutenção e a expansão da aplicação, permitindo a inclusão de novas telas, alteração na seqüência de telas, criação de caminhos alternativos de navegação em uma aplicação etc.

O *Model-View-Controller* (*MVC*) orienta a organização do código, definindo responsabilidades para cada componente da aplicação. É muito comum confundir o *MVC* com o que se convencionou chamar de *Programação em Três Camadas*, que propõe uma divisão da aplicação em componentes de *Apresentação*, *Negócios* e *Persistência*. Fazendo um paralelo entre estes dois padrões de projeto, o componente *Modelo* do *MVC* abrange as camadas de *Negócio* e *Persistência*, e a camada de *Apresentação* incorpora os componentes *Vista* e *Controlador* do *MVC* (figura 4.2).

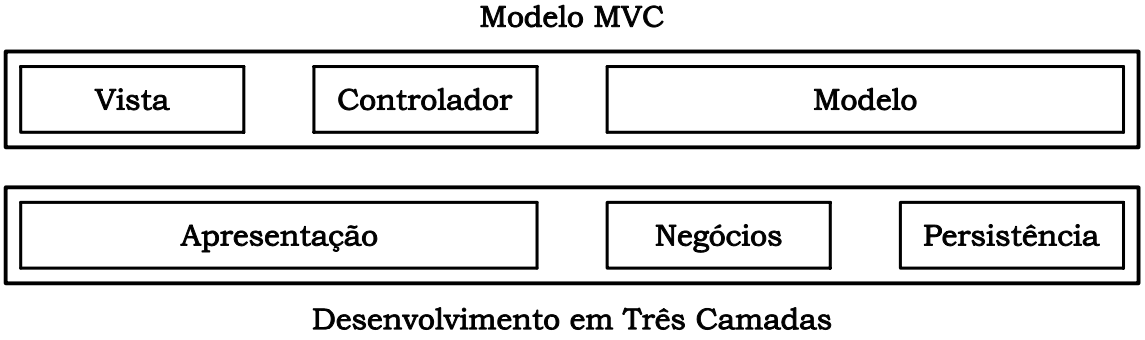


Figura 4.2: Componentes dos padrões MVC e Três Camadas

O uso dos padrões MVC e a Programação em Três Camadas é vantajoso na maioria das aplicações. A fusão dos dois padrões gera uma *Programação em Quatro Camadas*: *vista*, *controlador*, *negócios* e *persistência* (figura 4.3). Essa divisão em camadas deve nortear a

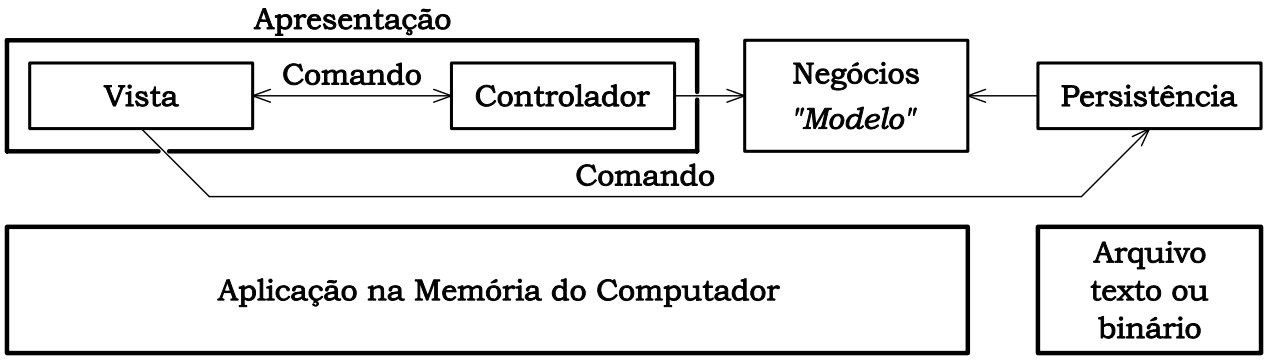


Figura 4.3: Programação em Quatro Camadas

forma como se constrói o código (Lozano 2004).

Na parte superior da figura 4.3 pode-se ver as quatro camadas lógicas da aplicação. A camada de *Negócios*, objetos de negócio, representa entidades tangíveis, em uma aplicação, as quais os usuários podem criar, acessar e manipular, enquanto realizam um *Caso de Uso* (seção 4.4 na página 39). Os objetos de negócio possuem tipicamente estado, são persistentes e têm vida longa. Eles contém dados e modelam o comportamento do negócio (Gupta 2004). Optou-se denominar a camada de *Negócios* por *Modelo*, mais adequado neste caso. Na parte inferior da figura 4.3 observam-se as camadas físicas (somente duas, nesta versão da aplicação): arquivos XML ou objetos Java persistidos em disco e toda a lógica da aplicação na memória do computador.

A figura 4.3 também mostra a inclusão do padrão *Comando* na aplicação. Este padrão foi introduzido de maneira a estabelecer a comunicação entre as camadas *vista* e *controlador* e *vista* e *persistência*.

4.2 Requisitos do Sistema

A aplicação tem por finalidade o pré-processamento de análises via modelos discretos, reduzindo o esforço requerido na definição e verificação dos dados de geometria, topologia, condições de contorno, propriedades de material e carregamentos.

Nessa primeira versão disponibilizou-se um modelador que gera malhas bidimensionais de elementos finitos através de mapeamentos transfinitos.

Desejou-se que o modelo a ser implementado apresentasse as seguintes características:

1. Necessidade de um mínimo de dados de entrada;
2. Possibilidade do controle da densidade da malha;
3. Modelagem precisa do contorno;
4. Geração automática da topologia dos elementos;
5. Eficiência computacional;

6. Garantia de qualidade da malha;

7. Possibilidade de visualização e interferência do usuário a cada etapa.

4.3 Funções do Sistema

Uma função do sistema é uma atividade cuja responsabilidade é atribuída ao sistema. Estas funções serão traduzidas, na fase de projeto (capítulo 4), em métodos ou procedimentos que serão executados pelos componentes do sistema.

As funções podem ser categorizadas em evidentes, ocultas ou decorativas. As evidentes devem ser executadas pelo sistema e o usuário deve tomar conhecimento de sua execução. As ocultas devem ser executadas pelo sistema, sem torná-las visíveis para o usuário. As decorativas são opcionais, sua adição não afeta significativamente outras funções (Larman 2000).

A tabela 4.1 apresenta uma lista das principais funções do sistema.

Tabela 4.1: Principais funções do sistema

Função	**Categoria**
Criar um novo modelo	evidente
Salvar um modelo	evidente
Abrir um modelo anteriormente salvo	evidente
Fechar um modelo aberto	evidente
Imprimir a representação do modelo	evidente
Persistir o modelo como objeto Java para o sistema `INSANE`	evidente
Persistir o modelo como arquivo XML para aplicações do MEF	evidente
Manter listas de comandos executados para possibilitar a operação desfazer	oculta
Armazenar o último comando desfeito para possibilitar a operação refazer	oculta
Permitir a operação desfazer quando possível	evidente
Permitir a operação refazer quando possível	evidente
Possibilitar a indicação de objetos do modelo através da seleção de suas representações gráficas	evidente

continua na próxima página

Tabela 4.1: Principais funções do sistema (continuação)

Função	**Categoria**
Manter listas de objetos indicados para possibilitar a operação remover	oculta
Permitir a operação remover objetos selecionados	evidente
Possibilitar a visualização da representação gráfica do modelo	evidente
Possibilitar a visualização de uma área selecionada da representação do modelo	evidente
Possibilitar o aumento ou redução da representação gráfica do modelo	evidente
Possibilitar a visualização da área definida para o modelo	evidente
Possibilitar a visualização da área ocupada pela representação do modelo	evidente
Manter listas com os limites das áreas apresentadas pelo aplicativo	oculta
Permitir a visualização das áreas limitadas pelos valores armazenados nas listas	evidente
Redesenhar a representação gráfica do modelo quando ordenado	evidente
Permitir ajustes na representação gráfica do modelo	evidente
Exibir mensagens informativas para o usuário	decorativa
Possibilitar o aumento ou redução da representação de componentes do modelo	decorativa
Apresentar informações sobre o aplicativo e o sistema do qual faz parte	decorativa
Possibilitar a criação da primitiva ponto	evidente
Possibilitar a criação da primitiva segmento de reta	evidente
Possibilitar a criação da primitiva curva quadrática	evidente
Possibilitar a criação da primitiva curva cúbica	evidente
Possibilitar a criação de curvas (conjunto de primitivas)	evidente
Possibilitar a criação de sub-regiões (definidas por 2, 3 ou 4 curvas)	evidente
Possibilitar a divisão das primitivas	evidente
Possibilitar a geração de malhas	evidente
Permitir a atribuição de forças nodais	evidente
Permitir a atribuição de deslocamentos nodais prescritos	evidente
Permitir a atribuição de restrições aos deslocamentos nodais	evidente
Permitir a criação de apoios elásticos	evidente
Permitir a alteração das propriedades dos elementos criados	evidente
Permitir a criação de materiais para os elementos	evidente
Permitir a criação de forças de superfície	evidente
Permitir a criação de forças de corpo	evidente

4.4 Casos de Uso

Um *caso de uso* é um documento narrativo que descreve a sequência de eventos de um ator (um agente externo) que usa um sistema para completar um processo (Jacobson, Christerson, Jonsson & Övergaard 1992) apud (Larman 2000). Ele descreve, do início ao fim, um processo relativamente grande, que inclui, tipicamente, muitos passos ou transações. Casos de uso são histórias ou casos de utilização de um sistema, eles não são exatamente especificação de requisitos ou especificação funcional, mas ilustram e esclarecem requisitos na história que contam. Geralmente ele não é um passo ou atividade individual em um processo. É muito comum definir, inapropriadamente, um caso de uso como passos individuais, operações ou transações de um processo mais amplo.

Um *ator* é uma entidade externa ao sistema que, de alguma maneira, participa da história do caso de uso. Um ator, tipicamente, estimula o sistema com eventos de entrada ou recebe algo do mesmo. Os atores são representados pelo papel que eles desempenham no caso de uso (Larman 2000).

Os símbolos UML utilizados para representar um caso de uso e um ator são apresentados na figura 4.4.

A figura 4.5 ilustra um *Diagrama de Caso de Uso* para o aplicativo apresentado neste trabalho. A finalidade deste diagrama é apresentar os atores, casos de uso e os relacionamentos entre eles.

(a) caso de uso (b) ator

Figura 4.4: Símbolos UML

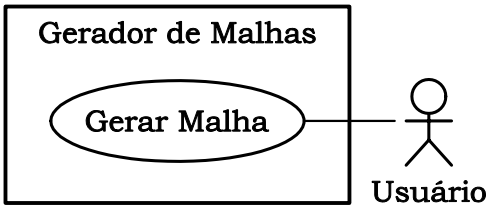


Figura 4.5: Diagrama de Caso de Uso

Um caso de uso possui um *Cabeçalho*, uma *Seqüência Típica de Eventos* e uma *Seqüência Alternativa de Eventos*. A *Seqüência Típica de Eventos* descreve com detalhes a interação entre os atores e o sistema - a história normal das atividades e do término bem-sucedido de um processo. Situações alternativas não são incluídas na seqüência típica. A *Seqüência*

Alternativa de Eventos descreve alternativas importantes ou exceções que podem surgir em relação à seqüência típica. Se as alternativas forem complexas, elas podem ser expandidas como casos de usos.

A seguir apresenta-se o caso de uso desenvolvido neste trabalho. Esse caso de uso promove uma fácil compreensão dos processos e requisitos do sistema.

Cabeçalho

Caso de uso:	Gerar malha
Ator:	Usuário do sistema
Visão Geral:	Um usuário executa o sistema, gera uma malha para uma região definida e obtém como resultado um arquivo XML ou binário (objeto Java) que será o arquivo de entrada para uma aplicação do Método dos Elementos Finitos.

Seqüência Típica de Eventos

Ação do Ator	Resposta do Sistema
1. Este caso de uso começa quando o usuário inicia a interação com o sistema para gerar os dados de entrada para uma aplicação do MEF.	
2. O usuário aciona um botão associado à criação de um *novo modelo*.	**3.** O sistema instancia um objeto que representa o *modelo*, prepara a interface para a primeira atividade (criar as primitivas que vão definir o contorno de cada sub-região) e instancia um controlador adequado para receber as próximas entradas do usuário.

continua na próxima página

Seqüência Típica de Eventos (continuação)

Ação do Ator	Resposta do Sistema
4. O usuário aciona os botões associados à criação das primitivas (pontos, segmentos de retas, curvas quadráticas e cúbicas) e comanda a criação, uma a uma, de todas as primitivas necessárias para definir o contorno de cada sub-região do modelo.	**5.** O sistema adiciona ao modelo cada uma das primitivas criadas e apresenta na área de desenho uma representação para cada uma das primitivas (figura 4.6a). Simultaneamente, são apresentadas mensagens que destacam detalhes da criação de cada primitiva, quando a operação é possível.
6. O usuário aciona o gerenciador de atividades para indicar o fim da primeira atividade.	**7.** O sistema prepara a interface para a próxima atividade (definir o contorno de cada sub-região) e instancia um controlador apropriado para esta atividade.
8. O usuário seleciona um grupo de primitivas em seqüência e aciona o botão associado ao comando criar curva.	**9.** O sistema verifica se o conjunto de primitivas atende aos requisitos necessários para a formação de uma curva e, se possível, cria uma curva. As curvas criadas são adicionadas ao modelo e uma representação para cada curva é apresentada na área de desenho. Simultaneamente, são apresentadas mensagens que destacam detalhes da criação de cada curva, quando a operação é possível.
10. Após criar duas, três ou quatro curvas que definem o contorno de uma sub-região, o usuário aciona um botão associado à criação de uma sub-região.	**11.** O sistema verifica se as curvas criadas definem uma região e gera uma região, se for possível. O sistema adiciona ao modelo cada uma das sub-regiões criadas e apresenta na área de desenho uma representação para cada uma das sub-regiões (figura 4.6b). Simultaneamente, são apresentadas mensagens que destacam detalhes da criação de cada sub-região, quando a operação é possível.

continua na próxima página

Seqüência Típica de Eventos (continuação)

Ação do Ator	Resposta do Sistema
12. Após definir todas as sub-regiões o usuário aciona o gerenciador de atividades para indicar o término desta atividade.	**13.** O sistema prepara a interface para a próxima atividade (dividir cada primitiva em um determinado número de partes) e instancia um controlador adequado para esta atividade.
14. O usuário seleciona as primitivas e determina o número de divisões para cada primitiva.	**15.** O sistema calcula as posições intermediárias em cada primitiva, cria um nó para cada posição, definindo uma forma discreta para cada curva e apresenta sua representação na área de desenho (figura 4.6c).
16. O usuário aciona o gerenciador de atividades para indicar o término da atividade dividir primitivas.	**17.** O sistema prepara a interface para a próxima atividade (gerar malhas) e instancia um controlador adequado para esta atividade.
18. O usuário seleciona as sub-regiões criadas e comanda a geração de uma malha para cada sub-região. Para cada sub-região, o usuário determina o tipo de elemento que deve ser usado na criação de cada malha.	**19.** O sistema verifica a consistência da topologia das sub-regiões e, se for possível, gera as malhas. O sistema calcula a posição dos nós intermediários e a incidência de cada elemento que define a malha. Os elementos criados são adicionados ao modelo e uma representação de cada elemento é apresentado na área de desenho (figura 4.6d).
20. O usuário aciona o gerenciador de atividades para indicar o término desta atividade.	**21.** O sistema prepara a interface para a próxima atividade (gerar os atributos nodais) e instancia um controlador adequado para esta atividade.

continua na próxima página

Seqüência Típica de Eventos (continuação)

Ação do Ator	Resposta do Sistema
22. O usuário seleciona um ou vários nós e aciona um botão associado à criação dos atributos nodais.	**23.** O sistema altera os atributos dos nós selecionados e cria uma representação gráfica para cada atributo nodal modificado (figura 4.6e).
24. O usuário aciona o gerenciador de atividades para indicar o término de mais uma atividade.	**25.** O sistema prepara a interface para a próxima atividade (gerar os atributos dos elementos) e instancia um controlador adequado para esta atividade.
26. O usuário seleciona os elementos e aciona um botão associado à criação dos atributos dos elementos.	**27.** O sistema altera os atributos dos elementos selecionados e cria uma representação gráfica para cada atributo de elemento modificado (figura 4.6f).
28. Após obter a malha desejada, o usuário aciona um botão associado à geração do arquivo XML.	**29.** O sistema cria um arquivo no formato XML, acrescenta neste arquivo as informações contidas no modelo e salva o arquivo com o nome e local indicados pelo usuário.
30. O usuário aciona o botão associado ao comando salvar modelo.	**31.** O sistema salva o modelo como objeto (arquivo binário) com o nome e local indicados pelo usuário. O arquivo salvo poderá ser aberto por outro segmento do `INSANE` ou pelo próprio aplicativo, a fim de que modificações sejam feitas.
32. O usuário comanda o encerramento do aplicativo.	**33.** O sistema exibe uma mensagem de aviso, caso o modelo aberto não tenha sido salvo, salva o modelo, se o usuário desejar, fecha o modelo e encerra o aplicativo.

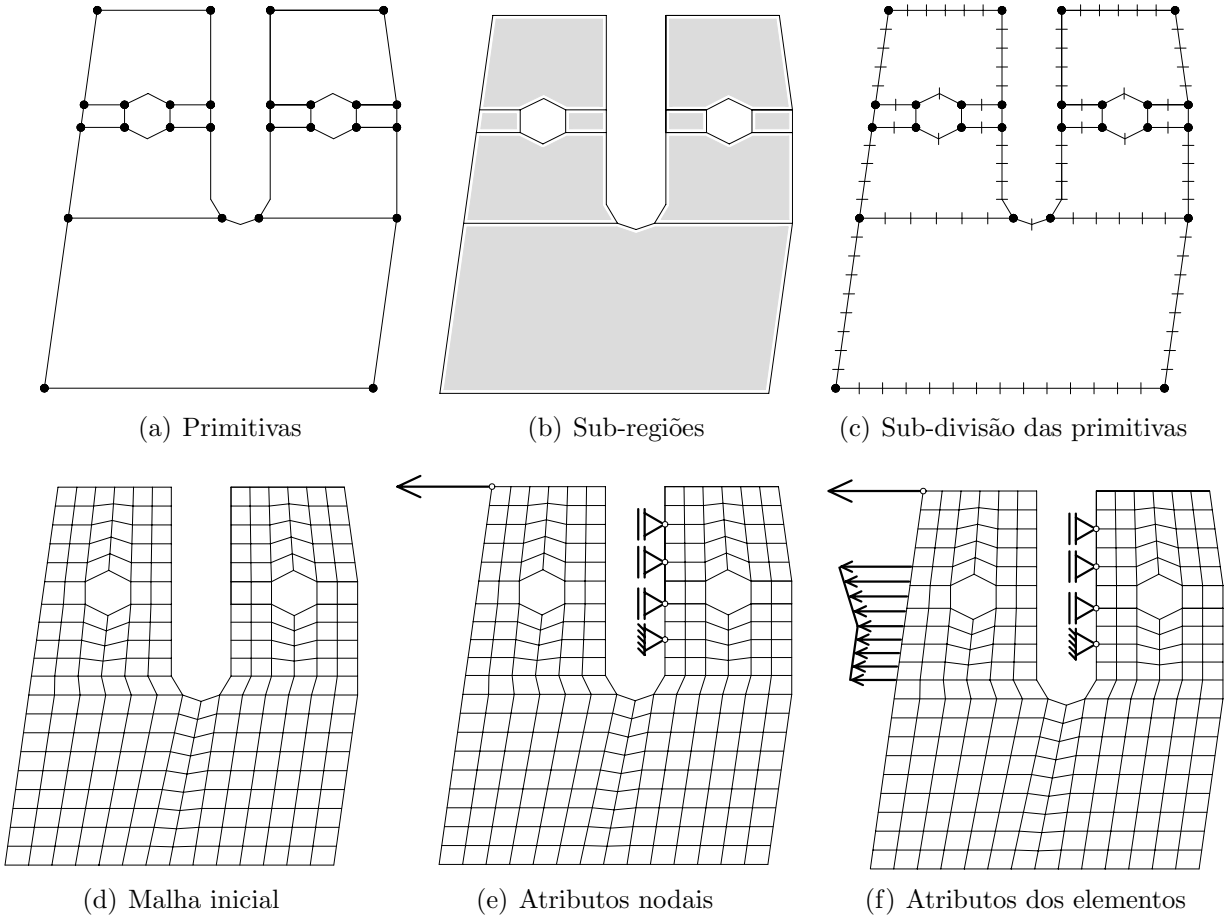
Seqüência Alternativa de Eventos

Item **5.**	O sistema apresenta mensagem informando porque a operação não é possível.
Item **9.**	O sistema apresenta mensagem informando porque a operação não é possível.

continua na próxima página

Seqüência Alternativa de Eventos (continuação)

Item **11.**	O sistema apresenta mensagem informando porque a operação não é possível.
Item **19.**	O sistema apresenta mensagem informando porque a operação não é possível.



(a) Primitivas (b) Sub-regiões (c) Sub-divisão das primitivas

(d) Malha inicial (e) Atributos nodais (f) Atributos dos elementos

Figura 4.6: Processo de geração da malha

4.5 Modelo Conceitual

Na *análise estruturada*, os recursos usados para a decomposição (divisão do problema em unidades compreensíveis) são os processos ou as *funções*. Porém, na *análise orientada a objetos*, tal decomposição é feita fundamentalmente por *conceitos*. Uma tarefa importante da fase de análise é, portanto, identificar diferentes conceitos no domínio do problema e documentar os resultados em um modelo conceitual.

Em UML, um modelo conceitual é exibido como um conjunto de diagramas de estrutura estática, nos quais não se definem operações. O termo *Modelo Conceitual* tem a vantagem de enfatizar fortemente os *conceitos* do domínio, as *associações* entre conceitos e os *atributos* de conceitos (Larman 2000).

Para a identificação dos *conceitos* utiliza-se como fonte de inspiração a seqüência típica de eventos do caso de uso em questão. Assim, os conceitos são as palavras-chave que surgem na descrição destes eventos. A notação UML para conceitos é a primeira seção de uma caixa (figura 4.7).

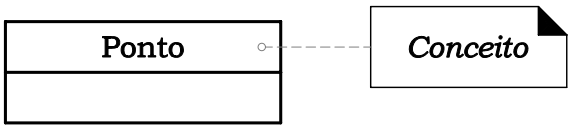


Figura 4.7: Símbolo UML para conceito

Uma *associação* é um relacionamento entre conceitos que indica uma conexão com significado e interesse. Em UML, elas são descritas como relacionamentos estruturais entre objetos de tipos diferentes.

Uma associação é representada como uma linha entre conceitos, com um nome (figura 4.8). A associação é inerentemente bidirecional, significando que é possível o percurso lógico dos objetos de um dos tipos para o(s) do(s) outro(s) tipo(s) e vice-versa.

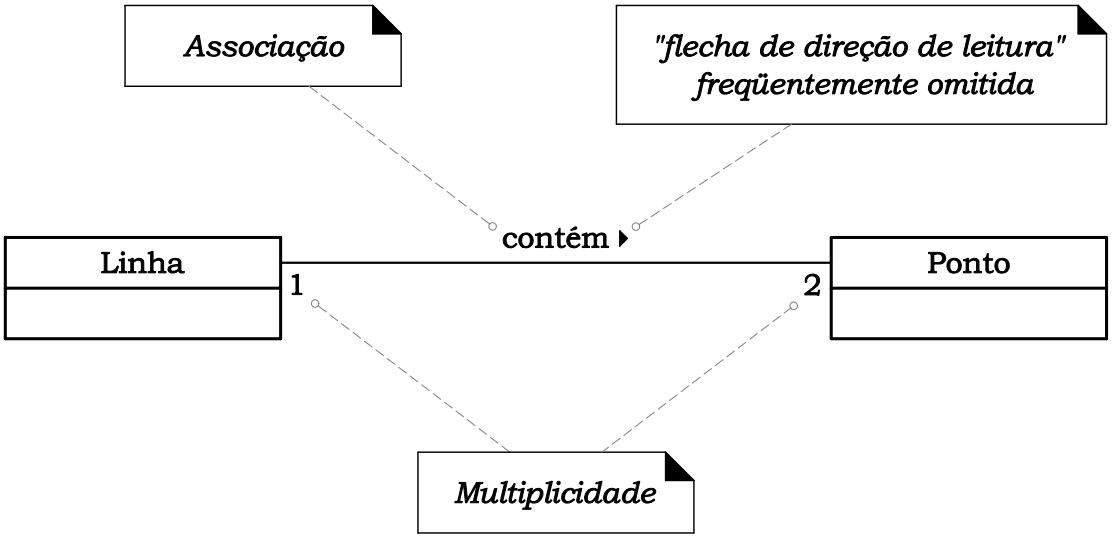


Figura 4.8: Representação UML de uma associação

Por convenção, lê-se a associação da esquerda para a direita ou de cima para baixo, embora UML não faça disso uma regra. Opcionalmente, pode-se colocar uma flecha de direção para indicar a direção de leitura do nome da associação (figura 4.8).

As pontas de uma associação podem conter uma expressão de multiplicidade (tabela 4.2), indicando o relacionamento numérico entre as instâncias dos conceitos (figura 4.8).

A multiplicidade define quantas instâncias de um tipo A podem estar associadas a uma instância de um tipo B. Alguns exemplos de expressões de multiplicidade são mostrados na tabela 4.2.

Tabela 4.2: Expressões de multiplicidade

Expressão	**Significado**
*	zero ou mais
1..*	um ou mais
1..13	um a treze
2	exatamente dois
9,14,19	exatamente nove, quatorze ou dezenove

Finalmente, para conceber um modelo conceitual é necessário definir os *atributos de conceitos*. Um atributo é um valor de dados lógico de um objeto. Em UML, os atributos são mostrados na segunda seção da caixa que contém o conceito(figura 4.9).

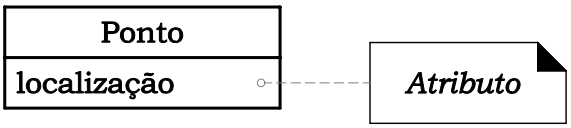


Figura 4.9: Símbolo UML para atributo de conceito

A figura 4.10 apresenta o modelo conceitual para o domínio do gerador de malhas do `INSANE`. Com este modelo, procura-se capturar as abstrações essenciais e as informações requeridas para compreender o domínio, no contexto dos requisitos presentes, auxiliando as pessoas na compreensão desse domínio - seus conceitos, sua terminologia e seus relacionamentos. As figuras 4.11 a 4.14 detalham o modelo conceitual da figura 4.10.

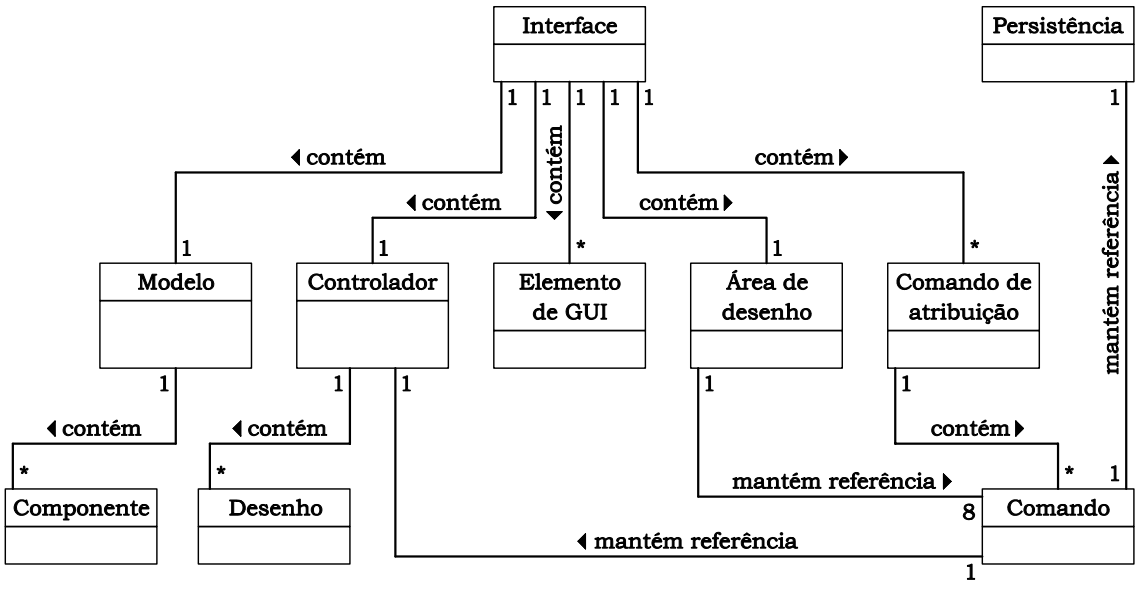


Figura 4.10: Modelo conceitual do gerador de malhas

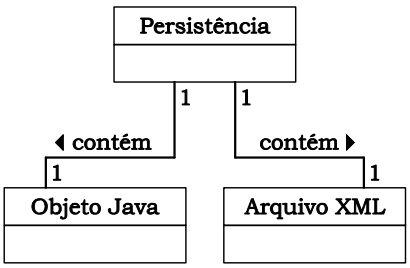


Figura 4.11: Modelo conceitual (detalhamento da Persistência)

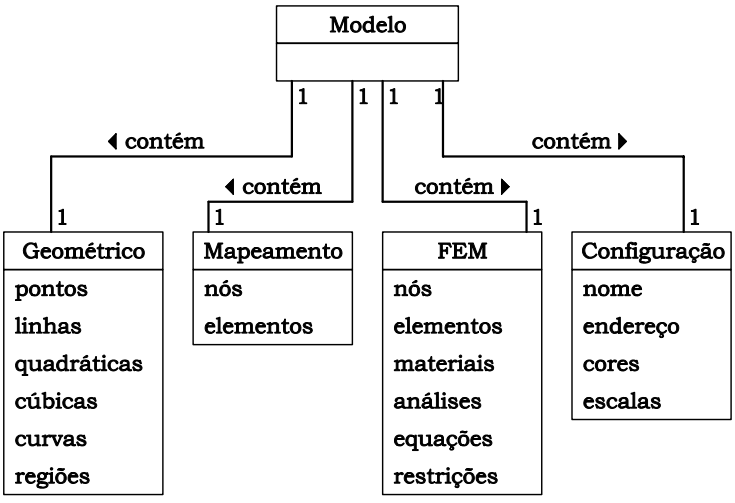


Figura 4.12: Modelo conceitual (detalhamento do Modelo)

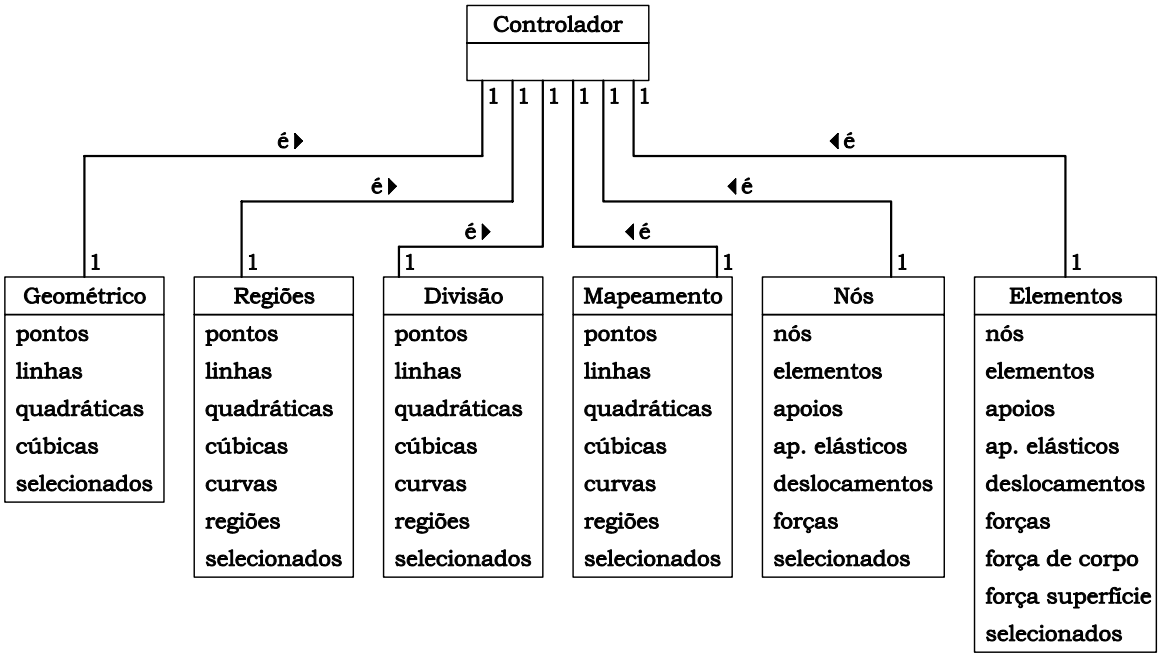


Figura 4.13: Modelo conceitual (detalhamento do Controlador)

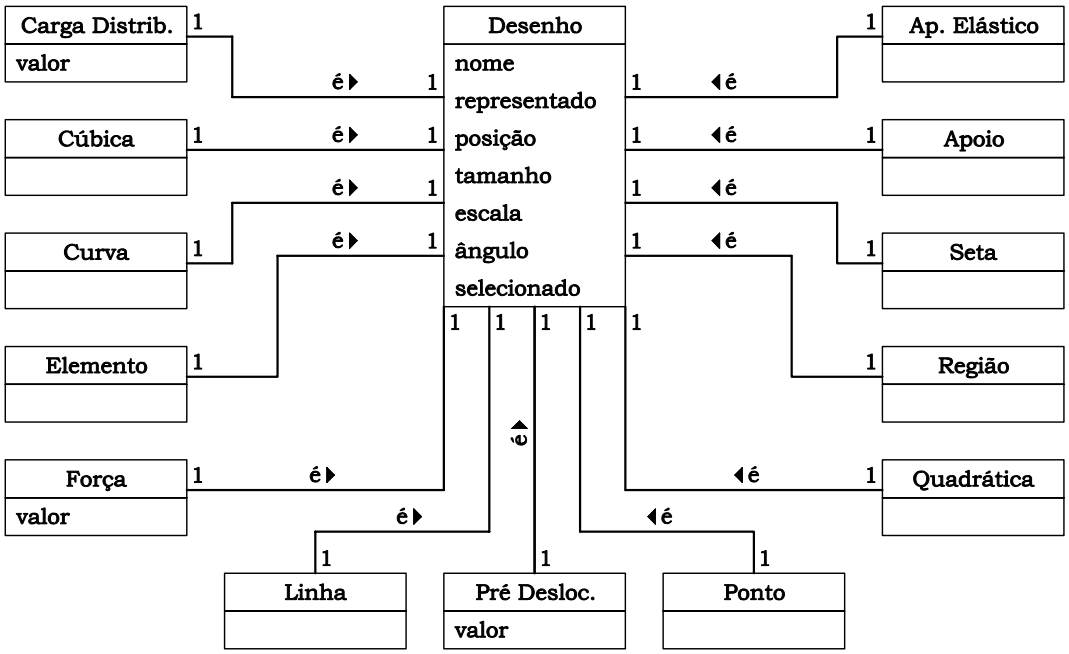
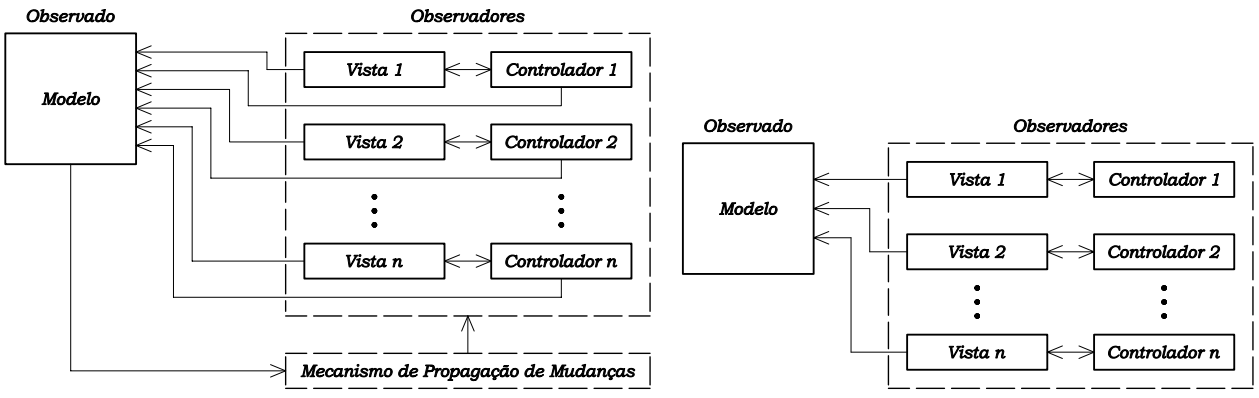


Figura 4.14: Modelo conceitual (detalhamento do Desenho)

4.6 Padrões de Projeto Utilizados

Na fase de análise foi estabelecida uma arquitetura, baseada no padrão MVC, em quatro camadas, a saber, *vista*, *controlador*, *modelo* e *persistência*. Estas camadas orientam a organização do código, definindo responsabilidades para cada componente da aplicação. O maior benefício desta arquitetura consiste na facilidade de manutenção e expansão do código (Lozano 2004). Entretanto, um efeito colateral comum resultante do particionamento de um sistema em uma coleção de classes cooperantes é a necessidade de manter a consistência entre objetos relacionados. Isso deve ser feito sem provocar um forte acoplamento entre as classes para não dificultar a reutilização das mesmas. O padrão de projeto *Observer* propõe a classificação dos objetos dependentes em observador (*observer*) e observado (*observable*) (Gamma et al. 1995). O objeto *observable* contém o núcleo dos dados e os objetos *observer* são dependentes desses dados.

O padrão *Observer* propõe a criação de um mecanismo de propagação de mudanças para garantir a consistência e a comunicação entre os componentes observadores (`Controlador` e `Vista`) e o componente observado (`Modelo`), como mostra a figura 4.15a. Quando um observador é criado, ele se registra em uma lista de objetos dependentes existente no mecanismo. Este é disparado quando ocorrem mudanças no objeto observado, e ele se encarrega de informar aos objetos observadores registrados para se atualizarem.



(a) Esquema geral (b) Esquema adotado

Figura 4.15: Componentes dos padrões `MVC` e `Observer`

O esquema apresentado na figura 4.15a é mais adequado para o caso geral, quando existem vistas múltiplas que simultaneamente apresentam dados do modelo. Na implementação do gerador de malhas foram usadas diversas vistas, havendo porém, apenas uma vista e um controlador ativos em cada atividade. Essa particularidade proporcionou a elaboração de um sistema de funcionamento mais simples (figura 4.15b). Como existe um único controlador ativo, e este transmite as requisições de serviços para o modelo, ele mesmo pode ficar encarregado de disparar o processo de atualização da vista, substituindo o *Mecanismo de Propagação de Mudanças* (figura 4.15a). A segunda modificação consiste em manter listas de desenhos referentes aos dados do modelo no controlador ativo, esse procedimento promove um maior isolamento entre o modelo e a vista uma vez que esta não precisa acessar o modelo para apresentar seus dados, passando a apresentar apenas as listas de desenhos contidas no controlador ativo.

O inter-relacionamento entre as camadas (observador e observado) é conseguido, principalmente, através da implementação do padrão de projeto de *software* denominado *Comando* (Grand 1998).

O padrão Comando (*Command*) encapsula uma função como um objeto. A super classe de *Command*, tipicamente, possui um único método com um nome do tipo *do*, *run* ou *perfom*. Uma instância de uma subclasse é criada sobreescrevendo este método, encapsulando também, normalmente, algum estado da classe. O comando pode então ser passado como um objeto, e executado invocando-se o método. As figuras 4.16, 4.17 e 4.18 ilustram o uso do comando no interrelacionamento entre camadas.

A figura 4.16 exemplifica este relacionamento para o caso da tarefa de adição de uma entidade geométrica ao modelo corrente e sua visualização. Como pode ser visto na figura, o fluxo de informações para realização de tal tarefa ocorre em quatro etapas. Na primeira etapa, o objeto `Comando`, responsável pela tarefa, aciona o `Controlador` ativo informando a requisição. A seguir (2), o `Controlador` cria o objeto correspondente à entidade geométrica e o adiciona ao `Modelo` pertinente. Na etapa 3, o `Controlador` cria objetos de desenho representativos dos objetos do `Modelo`. Finalmente, na etapa 4, os objetos de desenho pertencentes

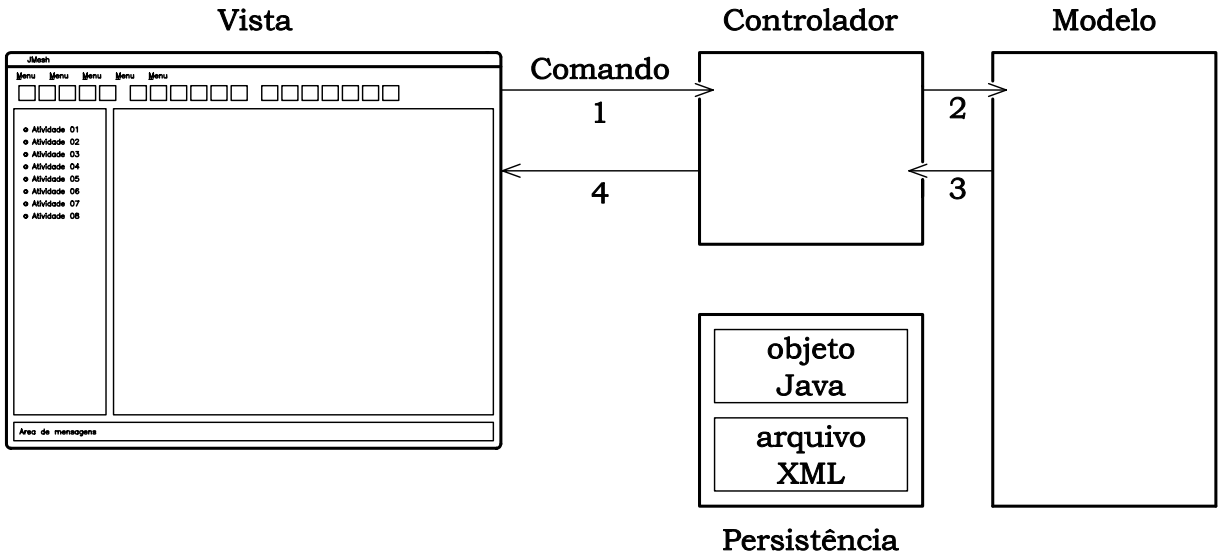


Figura 4.16: Relacionamento entre camadas para adição de uma entidade geométrica

ao `Controlador` são apresentados na área de desenho da `Vista`.

A figura 4.17 ilustra o relacionamento entre as camadas do aplicativo para executar a tarefa de *abrir um arquivo XML* que armazena os dados de uma malha. Para a realização de tal tarefa o processo foi dividido em cinco etapas principais. Na primeira etapa, o objeto `Comando`, responsável pela tarefa, acessa o arquivo XML. Na segunda etapa, o objeto `Comando` cria um objeto `Modelo` e objetos referentes às informações armazenadas no arquivo XML, e adiciona estes objetos ao `Modelo`. Na etapa seguinte (3), o `Comando` cria o objeto `Controlador`

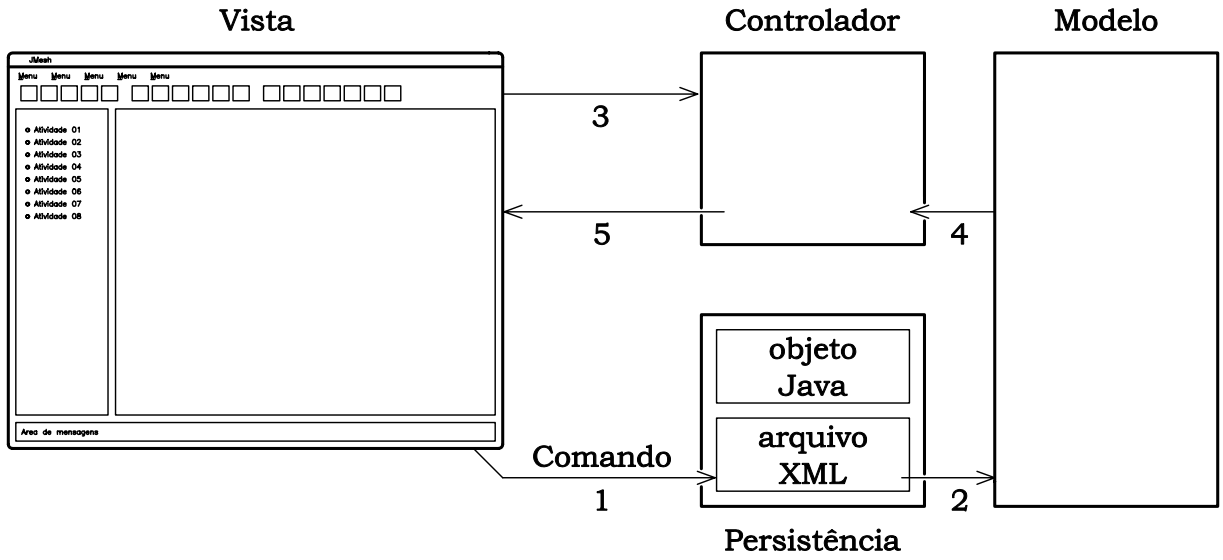


Figura 4.17: Relacionamento entre camadas para abrir um arquivo XML

apropriado. A seguir (4), o `Controlador` cria objetos de desenho representativos dos objetos do `Modelo`. Finalmente, na etapa 5, os objetos de desenho pertencentes ao `Controlador` são apresentados na área de desenho da `Vista`.

Como último exemplo do relacionamento entre as camadas do aplicativo, a figura 4.18 ilustra a tarefa *recuperar um* `Modelo` *persistido como objeto Java*. Para a realização de tal tarefa o processo foi dividido em cinco etapas principais. Na primeira etapa, o objeto `Comando`, responsável pela tarefa, acessa o arquivo binário e obtém o objeto `Modelo` (segunda etapa). Na etapa seguinte (3), o `Comando` cria o objeto `Controlador` apropriado. A seguir (4), o `Controlador` cria objetos de desenho representativos dos objetos do `Modelo`. Finalmente, na etapa 5, os objetos de desenho pertencentes ao `Controlador` são apresentados na área de desenho da `Vista`.

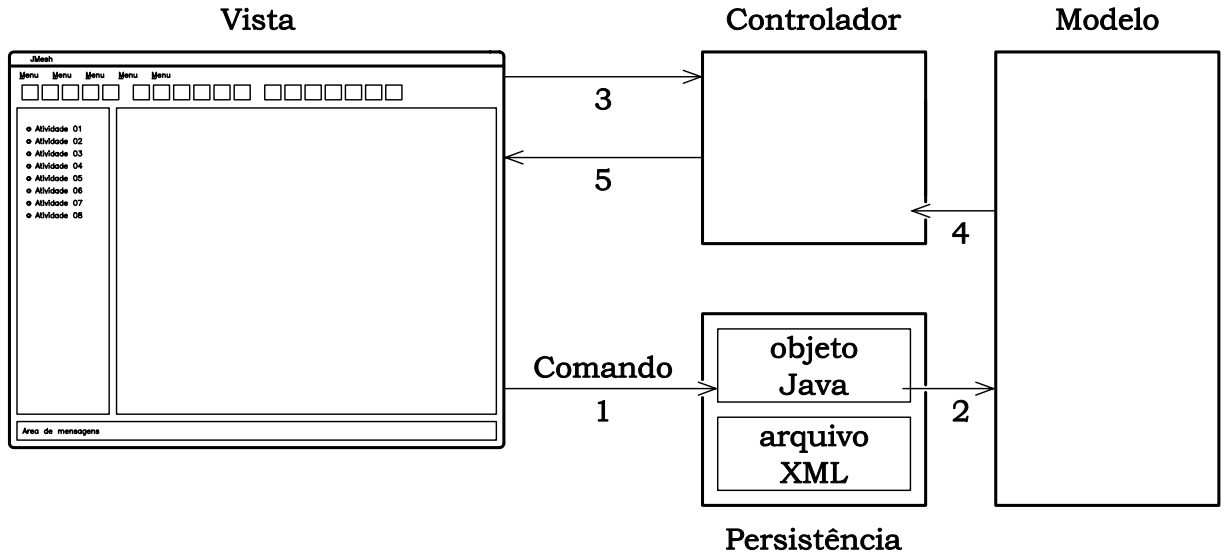


Figura 4.18: Relacionamento entre camadas para recuperar um `Modelo` persistido em arquivo binário

4.7 Diagramas de Classes

Em UML uma classe é representada por um retângulo dividido em três compartimentos: o compartimento de nome, contendo apenas o nome da classe modelada; o de atributos, contendo a relação dos atributos que a classe possui em sua estrutura interna; e o compartimento de

operações, contendo os métodos de manipulação de dados e de comunicação de uma classe com outras do sistema (figura 4.19).

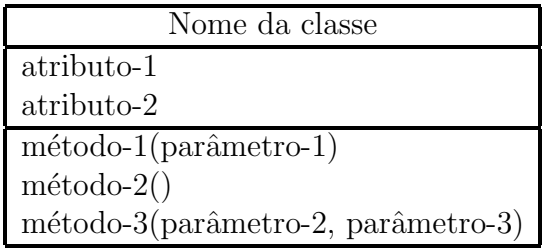
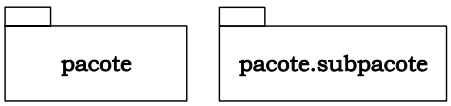


Figura 4.19: Símbolo UML para classe

Um diagrama de classes descreve as classes de objetos do sistema, seus atributos, operações e relacionamentos. O diagrama de classes é uma visão estática do sistema, porque a estrutura e o comportamento descritos são sempre válidos em qualquer ponto do ciclo de vida do sistema.

Para criar um diagrama de classes, as classes têm que estar identificadas, descritas e relacionadas entre si (Pagliosa 2004).

Um pacote (*package*) é uma coleção nomeada de classes (e possivelmente subpacotes). Os pacotes servem para agrupar classes relacionadas pelo objetivo, pelo escopo ou pela herança. Eles são delimitadores para as regras de controle de acesso a outras classes (Flanagan 2000). A figura 4.20 mostra o símbolo UML para um pacote.



(a) Pacote (b) Subpacote

Figura 4.20: Símbolos UML

O grande número de classes deste aplicativo impossibilita a apresentação das mesmas em um único diagrama. Assim, a apresentação das principais classes está dividida em *pacotes*. Esta divisão não interfere nos diagramas de herança, mas prejudica a compreensão dos diagramas de instância, pois estes podem incluir classes de pacotes distintos.

Para maior esclarecimento da instanciação entre as classes, detalhes dos atributos e métodos das principais classes da aplicação são apresentados no Manual de Desenvolvimento do Sistema (Brugiolo & Pitangueira 2004*a*).

As classes do gerador de malhas estão distribuidas nos pacotes mostrados na figura 4.21, cujos diagramas de herança e instância são apresentados a seguir.

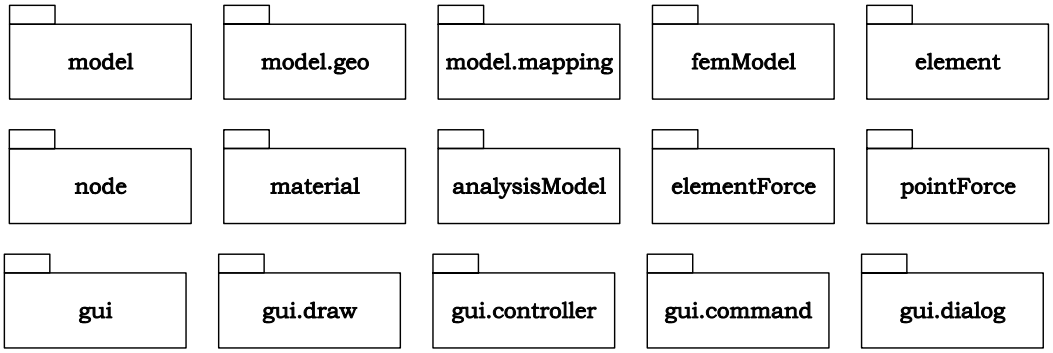


Figura 4.21: Pacotes do gerador de malhas

4.7.1 Pacote `model`

O pacote `model` contém a classe `Model`, que é composta por instâncias de classes que implementam a lógica da modelagem através do Método dos Elementos Finitos e por um objeto da classe `ModelState`, que armazena informações comuns a todas as classes de modelagem.

A figura 4.22 apresenta as classes do pacote `model` e as principais instâncias da classe

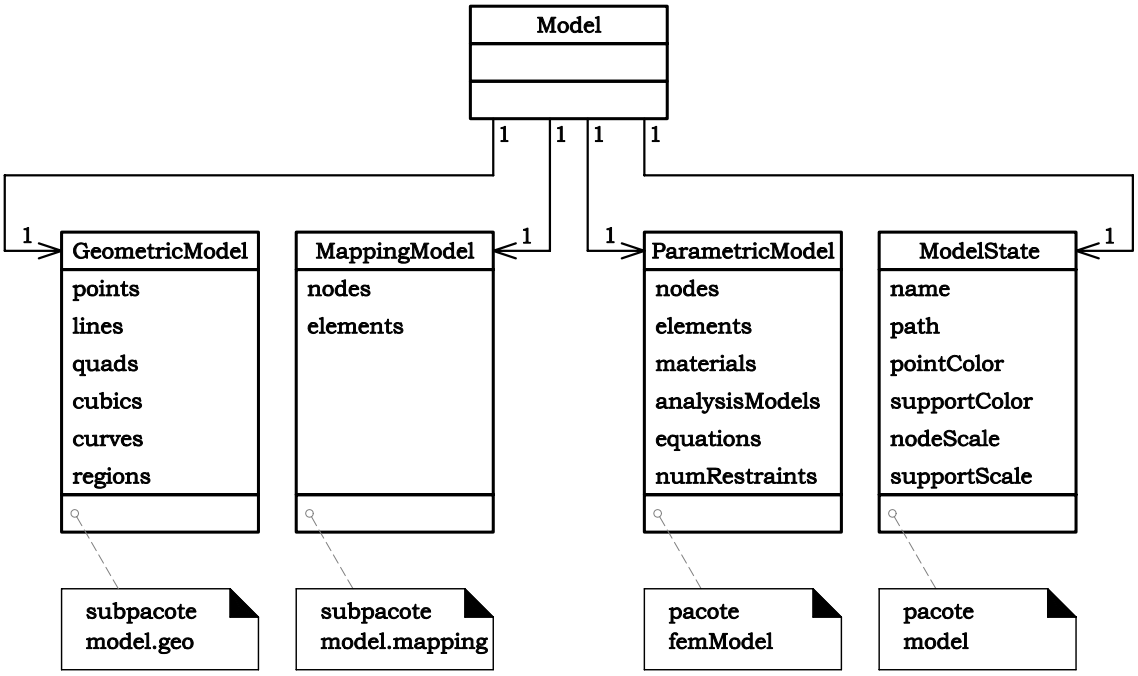


Figura 4.22: Diagrama de instâncias das classes do pacote `model`

`Model` que são detalhadas nas figuras seguintes.

Os subpacotes do pacote `model` agrupam classes responsáveis por etapas distintas do processo de modelagem. O subpacote `model.geometric` representa a etapa de modelagem geométrica, `model.mapping` representa a etapa de geração de malhas através de mapeamentos e `model.fem` a etapa de criação da discretização segundo o MEF.

A classe `GeometricModel` (do subpacote `model.geo`) é responsável por armazenar a estrutura de dados representativa da geometria do modelo. As demais classes do subpacote `model.geo`, apresentadas na figura 4.23, estão associadas às definições (ponto, segmento de

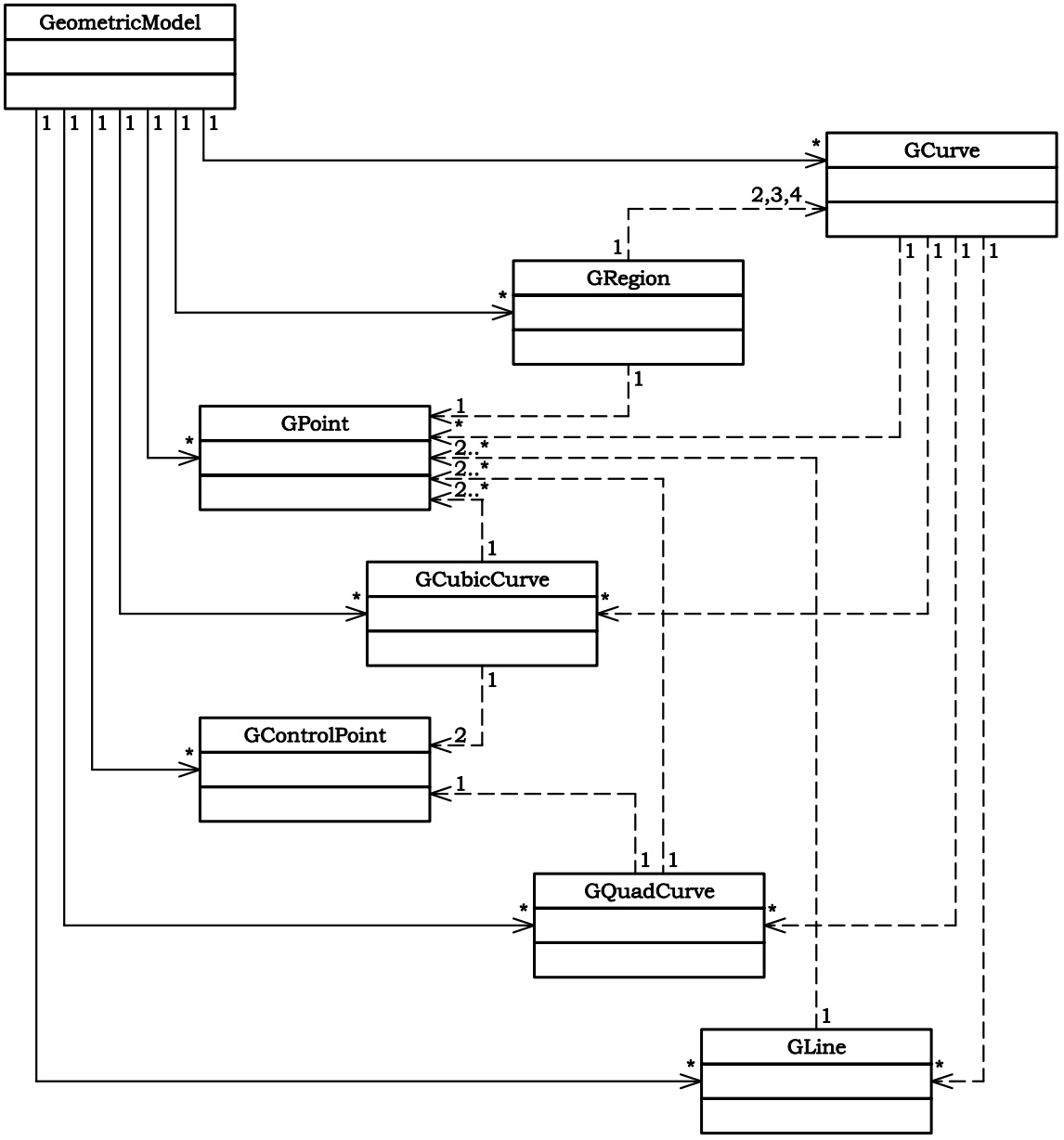


Figura 4.23: Diagrama de instâncias das classes do subpacote `model.geo`

reta, curva quadrática, cúbica, região e seu contorno) do modelo geométrico.

A classe `MappingModel` (do subpacote model.mapping) é responsável por armazenar os dados gerados palas classes de mapeamento (classe `Mapping`, figura 4.24). Os dados gerados pelas classes especializadas de `Mapping` consistem em objetos das classes `MapNode` e `MapElement`.

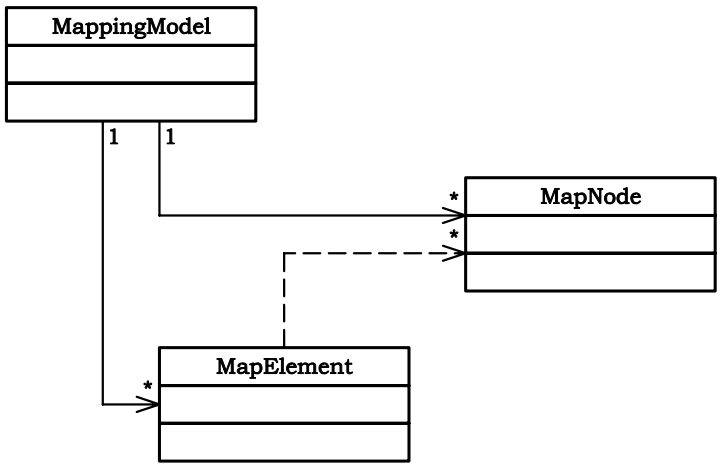


Figura 4.24: Diagrama de instâncias das classes do subpacote `model.mapping`

A figura 4.25 apresenta as classes do subpacote `mapping` e as relações de herança entre elas. Observa-se na figura que a classe responsável por gerar os nós e elementos (`Mapping`) possui apenas uma classe especializada (`Transfinite`), associada ao mapeamento transfinito, único tipo de mapeamento disponível na primeira versão do gerador de malhas.

A classe `Transfinite` possui três subclasses: `Lofting`, `Trilinear` e `Bilinear`, associadas, respectivamente, aos mapeamentos "lofting", trilinear e bilinear. As classes `Lofting` e `Bilinear` possuem classes mais especializadas que realizam o mapeamento com elementos quadrilaterais ou triangulares, lagrangeanos ou serendípticos. A classe `Trilinear` possui duas subclasses que realizam o mapeamento com elementos triangulares lagrangeanos ou serendípticos.

A classe `ParametricModel` (do subpacote femModel), especialização de `FemModel` (figura 4.26), representa o modelo do Método dos Elementos Finitos e possui objetos que representam todos os atributos de uma discretização baseada na formulação paramétrica deste método. A estrutura de dados da classe `ParametricModel` consiste, principalmente, de instâncias da classe `ParametricElement` (figura 4.27), cujos principais atributos são referências a objetos da

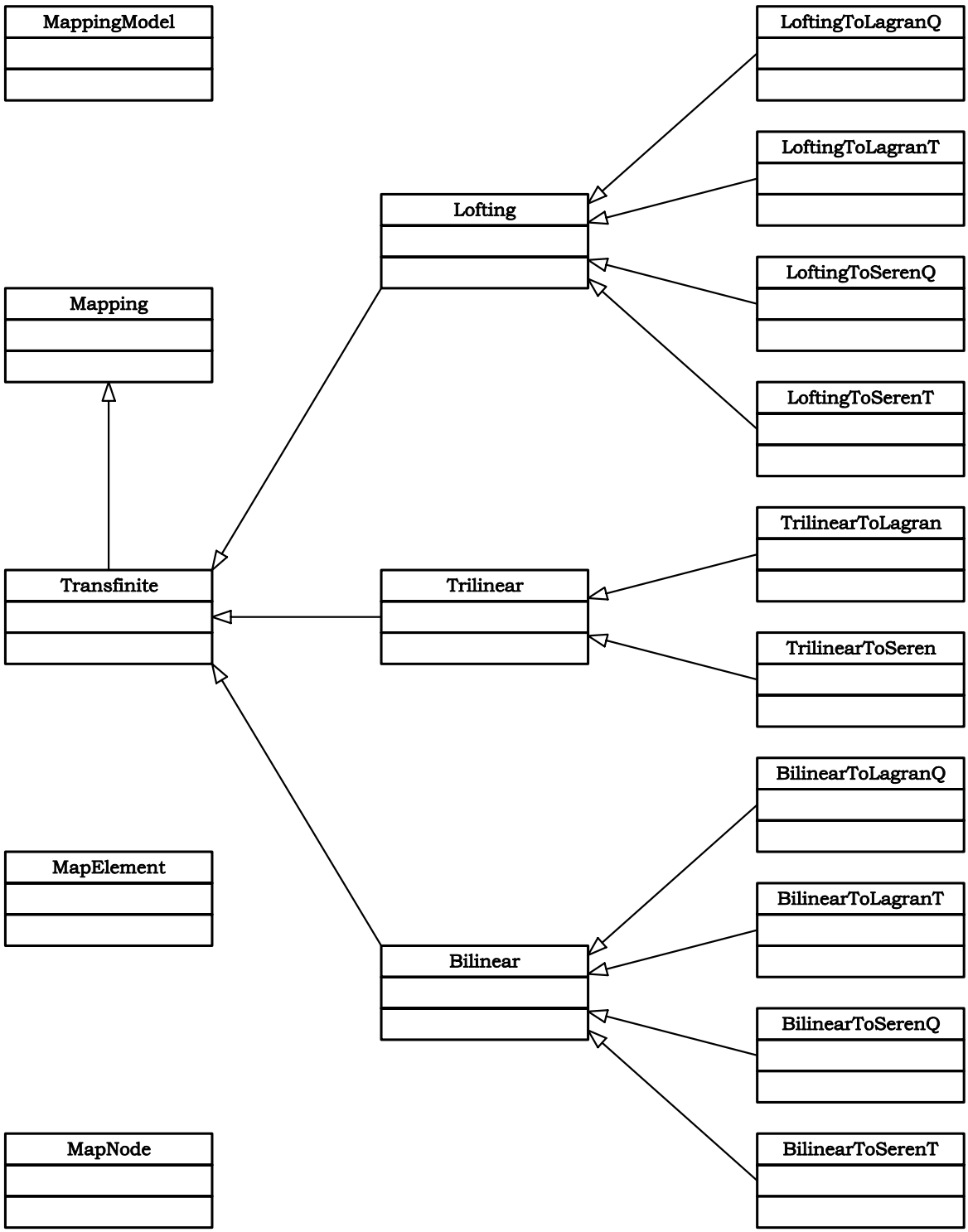


Figura 4.25: Diagrama de herança das classes do subpacote `model.mapping`

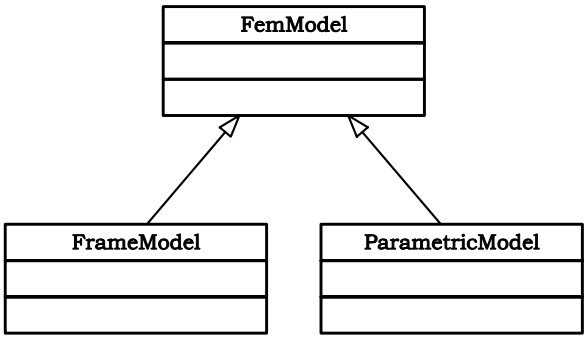


Figura 4.26: Diagrama de herança das classes do pacote `femModel`

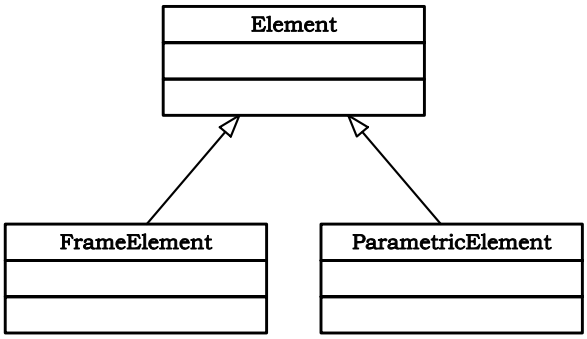


Figura 4.27: Diagrama de herança da classe `Element`

classe `Node` (sua incidência), a um objeto de uma classe especializada de `Material` (figura 4.28) e a um objeto de `AnalysisModel` (figura 4.29). Além destas referências, a classe `ParametricElement` possui ainda atributos relativos à ordem de integração numérica e espessura do elemento. Para caracterização das cargas, cada elemento possui uma lista de objetos da classe `PointForce` (figura 4.30) e três listas de objetos da classe `ElementForce` (figura 4.30).

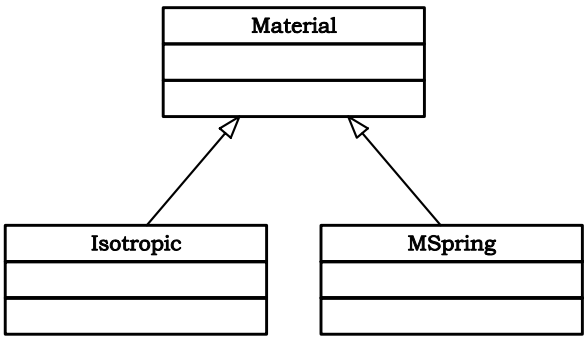


Figura 4.28: Diagrama de herança da classe `Material`

A lista de `PointForce` representa cargas pontuais não nodais aplicadas no elemento. As três listas de `ElementForce` representam cargas distribuídas ao longo de uma linha no elemento, ao longo de uma superfície no elemento e ao longo do volume do elemento. Os objetos

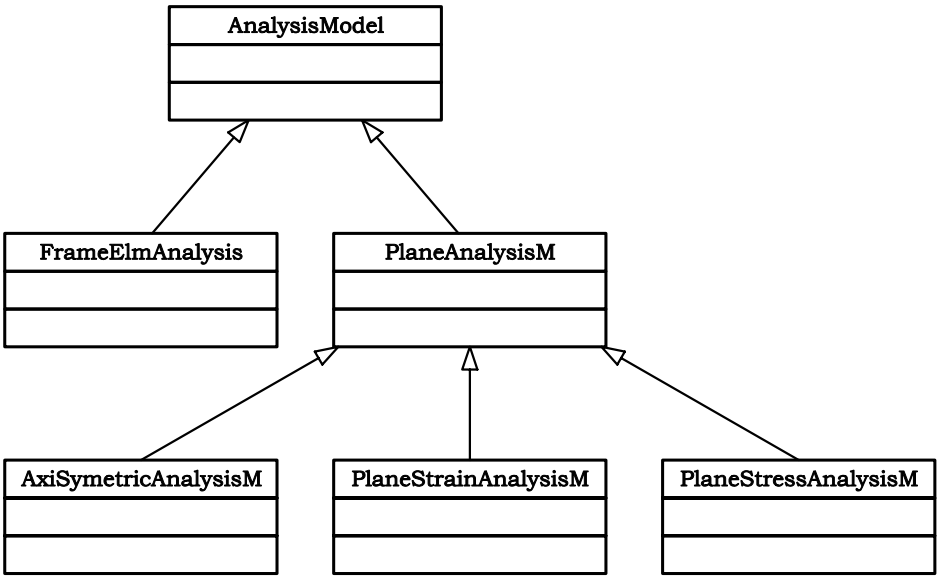


Figura 4.29: Diagrama de herança das classes do pacote `analysisModel`

destas listas são todos do tipo `ElementForce` porque tais carregamentos são descritos através de interpolação paramétrica de valores nodais prescritos dos mesmos.

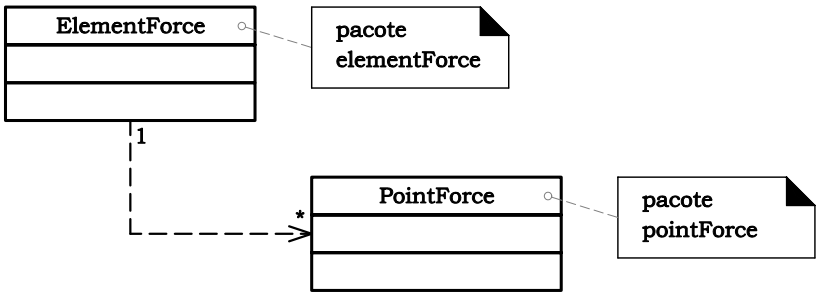


Figura 4.30: Diagrama de instâncias das classes `ElementForce` e `PointForce`

4.7.2 Pacote `gui`

O pacote `gui` e seus subpacotes agrupam as classes responsáveis pelas camadas de apresentação (camadas vista e controlador da figura 4.3 na página 35), bem como as classes que implementam as interações entre estas camadas.

A classe `Interface` (figura 4.31) possui um objeto da classe `Model`, um da classe `Controller`, um da classe `DrawingArea`, vários objetos da classe `Command` e vários objetos que compõem a interface gráfica com o usuário (GUI - "Graphical User Interface").

Os objetos `DrawingArea` e `Elemento de GUI` (figura 4.31) são intâncias de classes da *API*

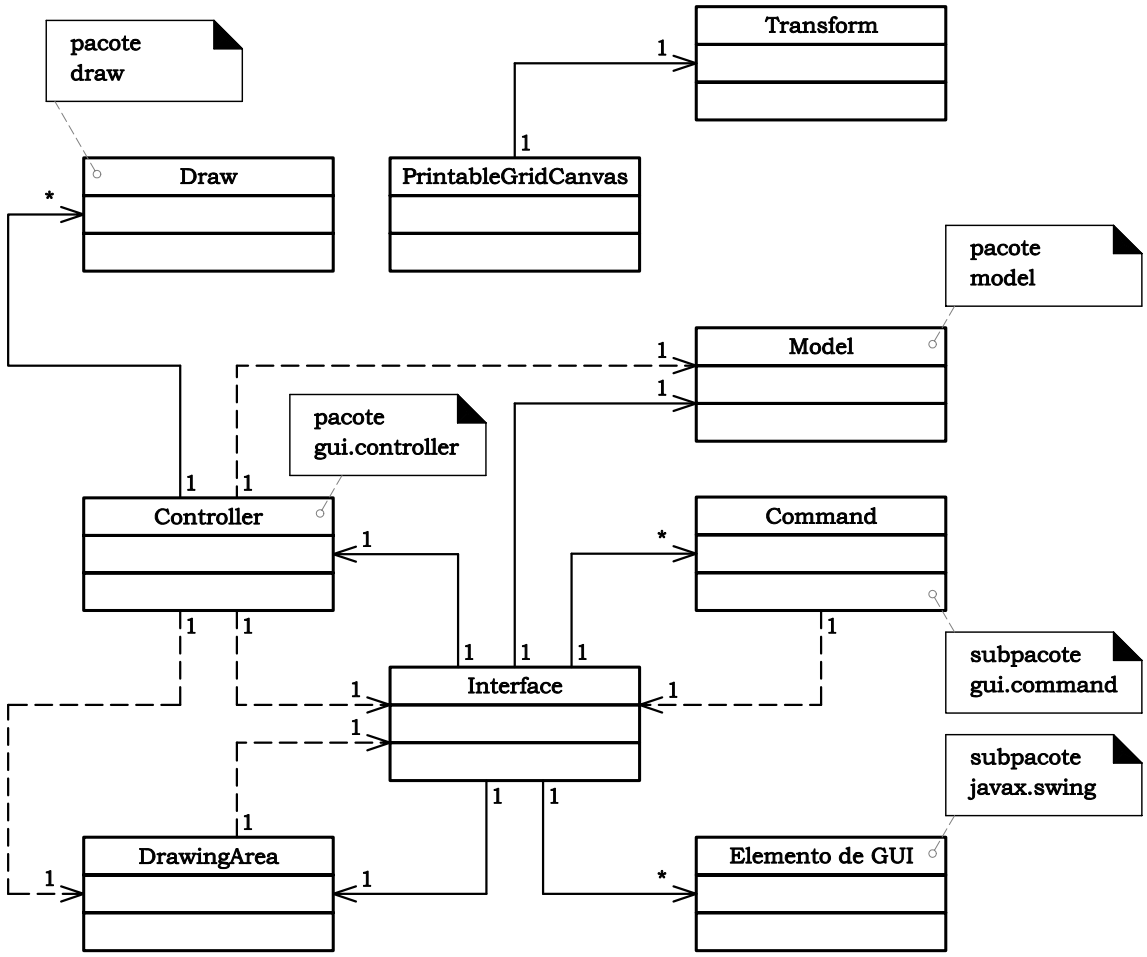


Figura 4.31: Diagrama de instâncias das classes do pacote `gui`

Java Swing (Horstmann & Cornell 2001*b*). O *Swing*, que faz parte da biblioteca *Java Foundation Classes*, oferece uma maneira de fornecer uma interface gráfica com o usuário em programas Java e de receber entradas do usuário com o teclado, o mouse ou outros dispositivos de entrada. Através do *Swing* pode-se criar aplicativos que apresentam uma interface gráfica com o usuário, utilizando-se os componentes: quadros, contêineres, botões, rótulos, campos de texto e áreas de texto, listas suspensas, caixas de verificação e botões de rádio. Estes componentes serão definidos no próximo capítulo, quando de sua utilização.

A classe `DrawingArea` é derivada da classe `PrintableGridCanvas` (figura 4.32) que é derivada de *JComponent*, que é superclasse para a maioria dos componentes de interface gráfica. A classe `PrintableGridCanvas` possui uma instância da classe `Transform` (figura 4.31) que é a classe responsável por fazer transformações entre os sistemas de coordenadas do dispositivo e do modelo.

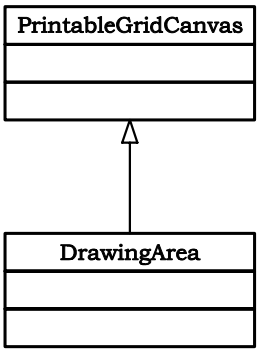


Figura 4.32: Diagrama de herança das classes `PrintableGridCanvas` e `DrawingArea`

O objeto `DrawingArea` herda de *JComponent*, um componente do *Swing*, onde um objeto *Graphics2D*, composto por intâncias de objetos de desenho do `Controller`, é desenhado.

As classes representando objeto de desenhos necessárias à visualização do processo de modelagem foram agrupados no subpacote `gui.draw`. O diagrama de herança destas classes estão mostradas na figura 4.33.

A *API Java2D* (*Application Programming Interface*) fornece recursos gráficos bidimensionais avançados que exigem manipulações gráficas complexas e detalhadas, muito amplos para serem trabalhados neste texto. O desenho com a *API Java2D* é realizado com uma instância da classe Graphics2D. A classe *Graphics2D* é uma subclasse de *Graphics*, portanto ela herda todos os recursos gráficos disponíveis na classe *Graphics*, que implementa métodos para desenho, manipulação de fontes, manipulação de cor, entre outros (Deitel & Deitel 2003).

A figura 4.34 apresenta as relações de instância entre as classes do subpacote `gui.draw`. A classe `CurveDraw` é responsável pela representação gráfica das curvas que definem as regiões, por isso possui referência às classes responsáveis pela representação gráfica das primitivas (que delimitam as regiões) e da seta, usada para indicar o sentido da mesma. A classe `ArrowLineDraw` é responsável pela representação gráfica do carregamento distribuído, sua representação reflete em suas relações de instância com as classes responsáveis pela representação gráfica de setas e retas.

A classe `Controller` (do subpacote `gui.controller`), como recomenda Grand (1998), é uma classe totalmente abstrata (em *Java*, uma interface) que faz referência ao `Model`, à `Interface` e à `DrawingArea`.

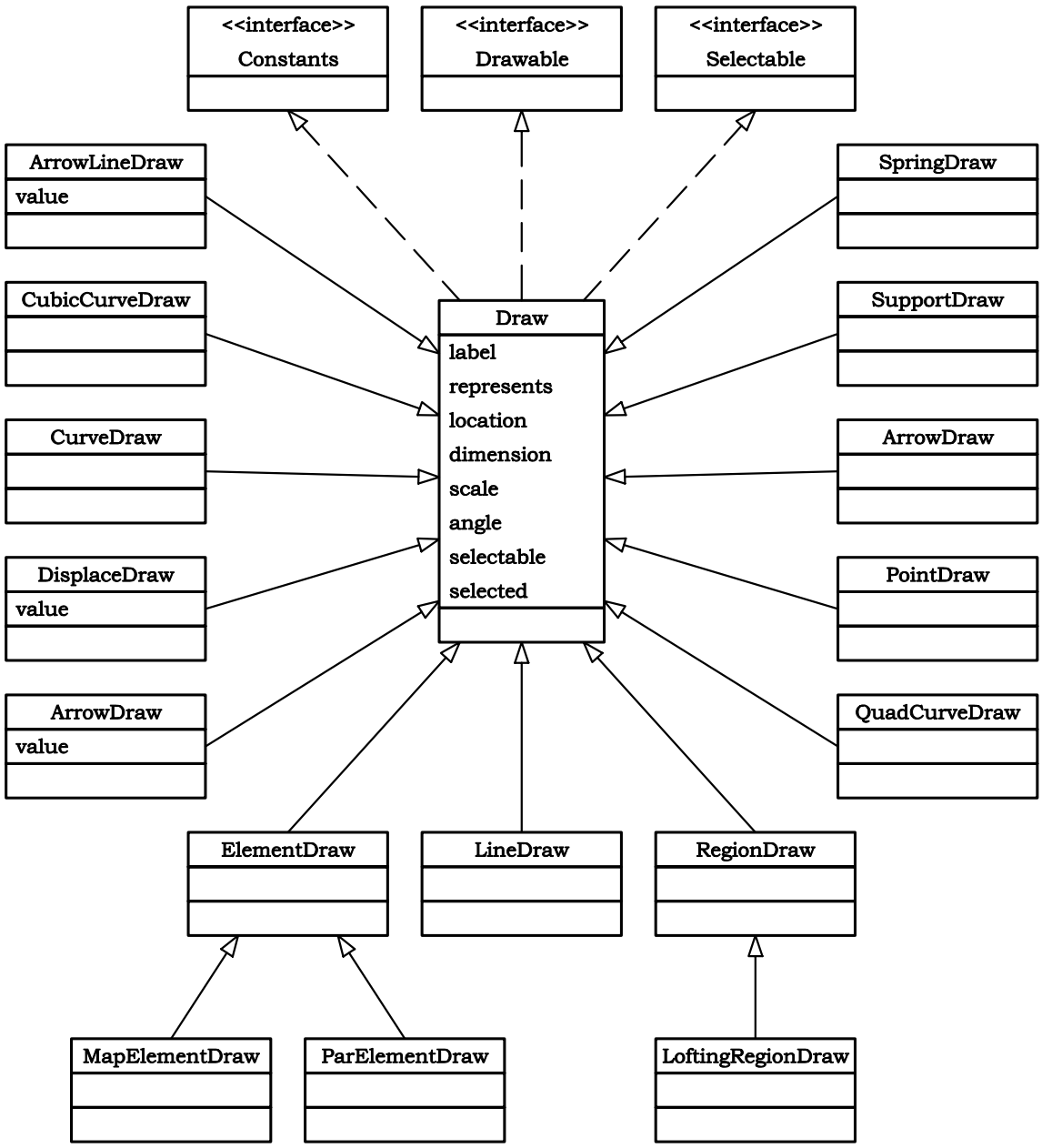


Figura 4.33: Diagrama de herança das classes do subpacote `gui.draw`

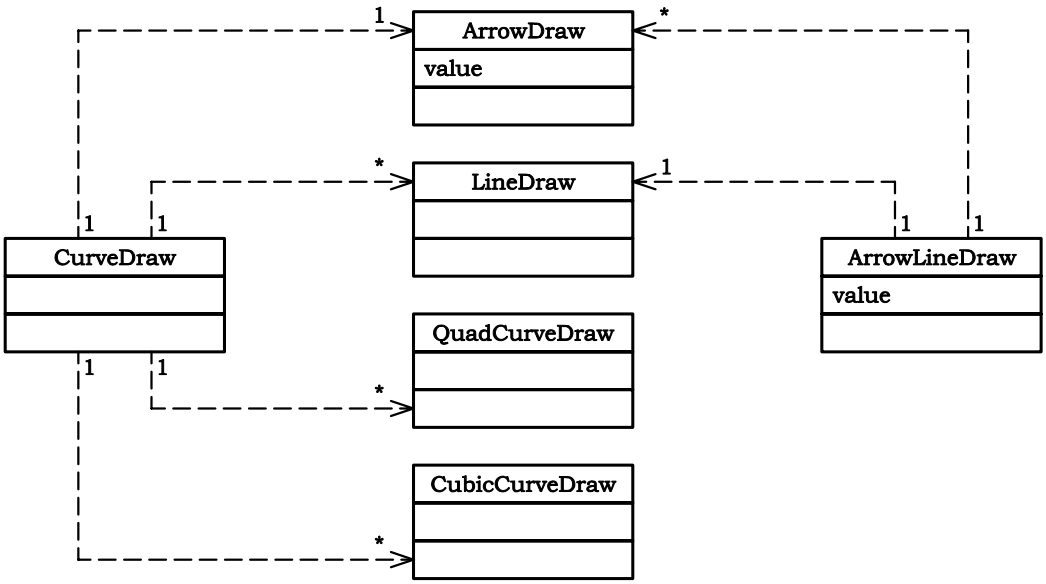


Figura 4.34: Diagrama de instâncias das classes do subpacote `gui.draw`

Cada uma das classes que implementam a interface `Controller` (figura 4.35) possui listas (instâncias de classes da *API Collections* (Horstmann & Cornell 2001*a*)) contendo instâncias de `Objetos de Desenho`, que são extensões de classes da *API* gráfica *Java2D* (Rowe 2001).

A *API Collections* oferece ao programador acesso a estruturas de dados pré-empacotadas e a algoritmos para manipular essas estruturas. As coleções são padronizadas de modo que os aplicativos possam compartilhá-las facilmente, sem se preocupar com detalhes de sua implementação. Essas coleções são escritas para ampla reutilização e ajustadas para rápida execução e utilização eficiente da memória.

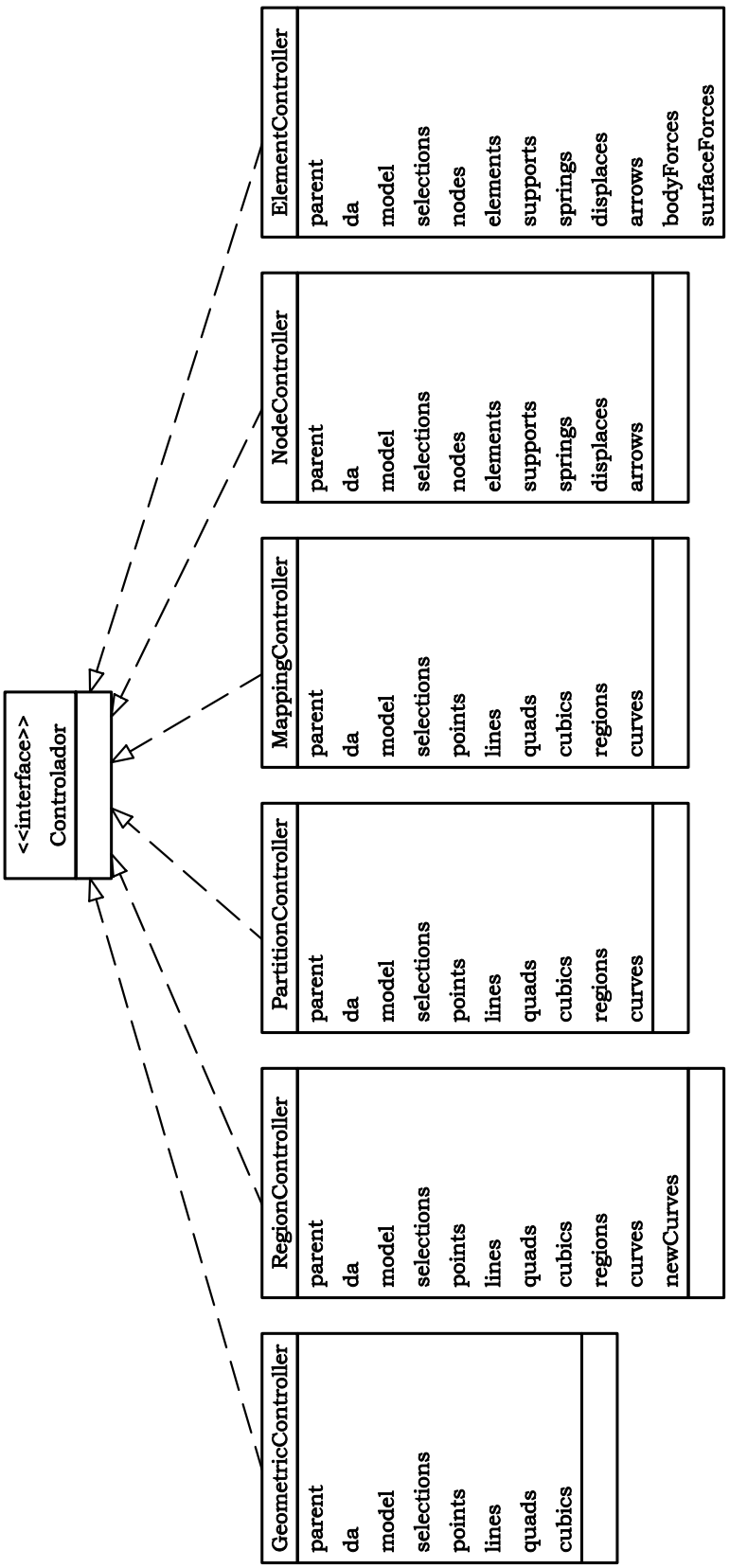


Figura 4.35: Diagrama de herança das classes do pacote `gui.controller`

Finalmente, as requisições que o usuário faz através dos elementos de interface gráfica (botões, menus, etc) foram tratadas através de classes que implementam a interface Command (figura 4.31), pertencentes ao subpacote `gui.command`. Cada subclasse de `Command` realiza uma operação específica e, por isso, apresenta grupos distintos de atributos. Elas são muito especializadas, o que dificulta a generalização que se observa nos diagramas anteriores. Nas figuras 4.36, 4.37, 4.38 e 4.39 são apresentadas várias subclasses de `Command`, observa-se um único tipo de relação de herança e relações de instância distintas.

Algumas classes, principalmente classes especializadas de `Command`, instanciam subclasses da interface `TabbedDialog` para possibilitarem a interação com o usuário. Essas subclasses apresentam diálogos específicos para recuperar entradas do usuário que são convertidos em parâmetros utilizados em suas rotinas. Estes diálogos formam o subpacote `gui.dialog`, apresentado na figura 4.40.

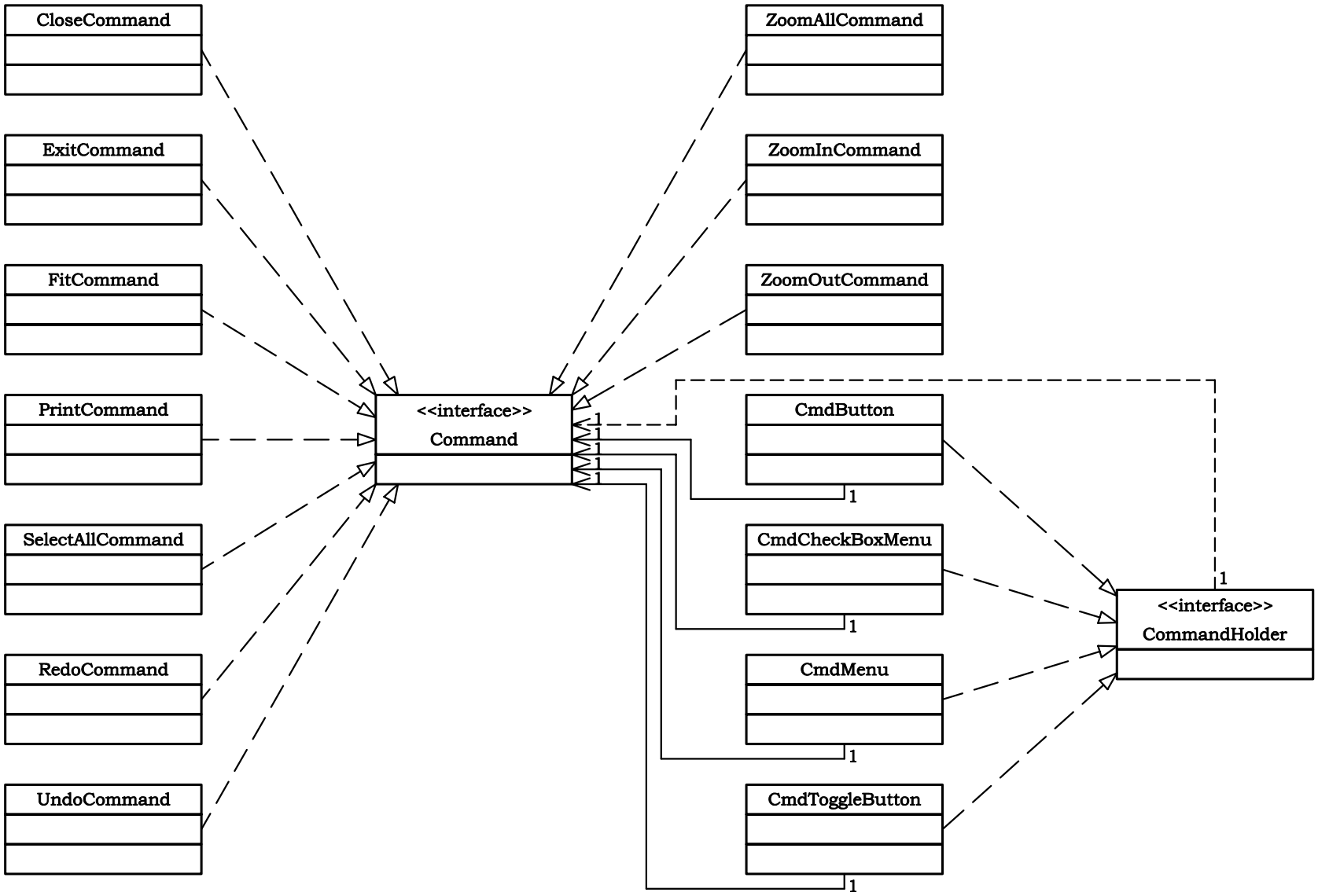


Figura 4.36: Diagrama de herança e instância das classes do subpacote `gui.command`

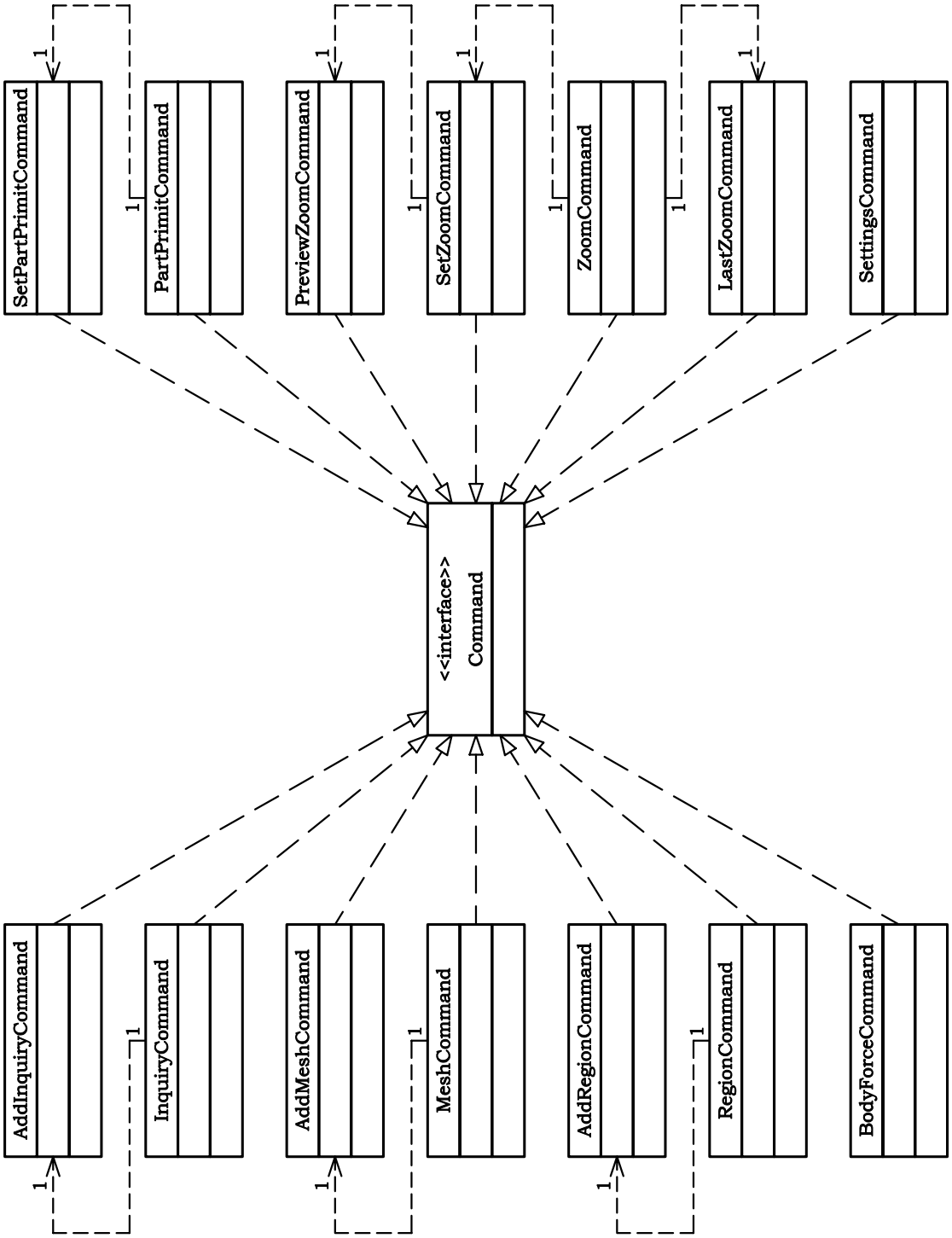


Figura 4.37: Diagrama de herança e instância das classes do subpacote `gui.command`

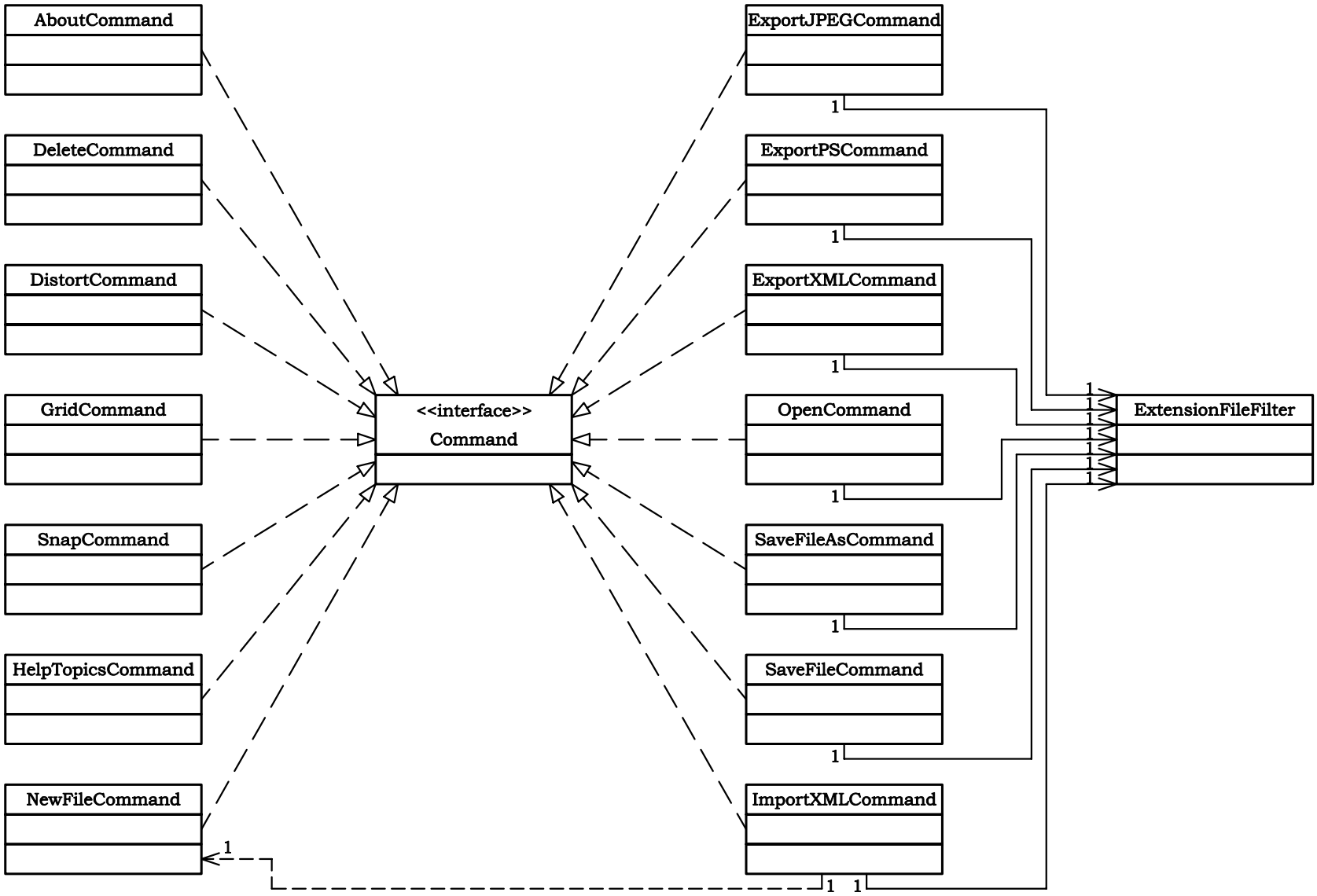


Figura 4.38: Diagrama de herança e instância das classes do subpacote `gui.command`

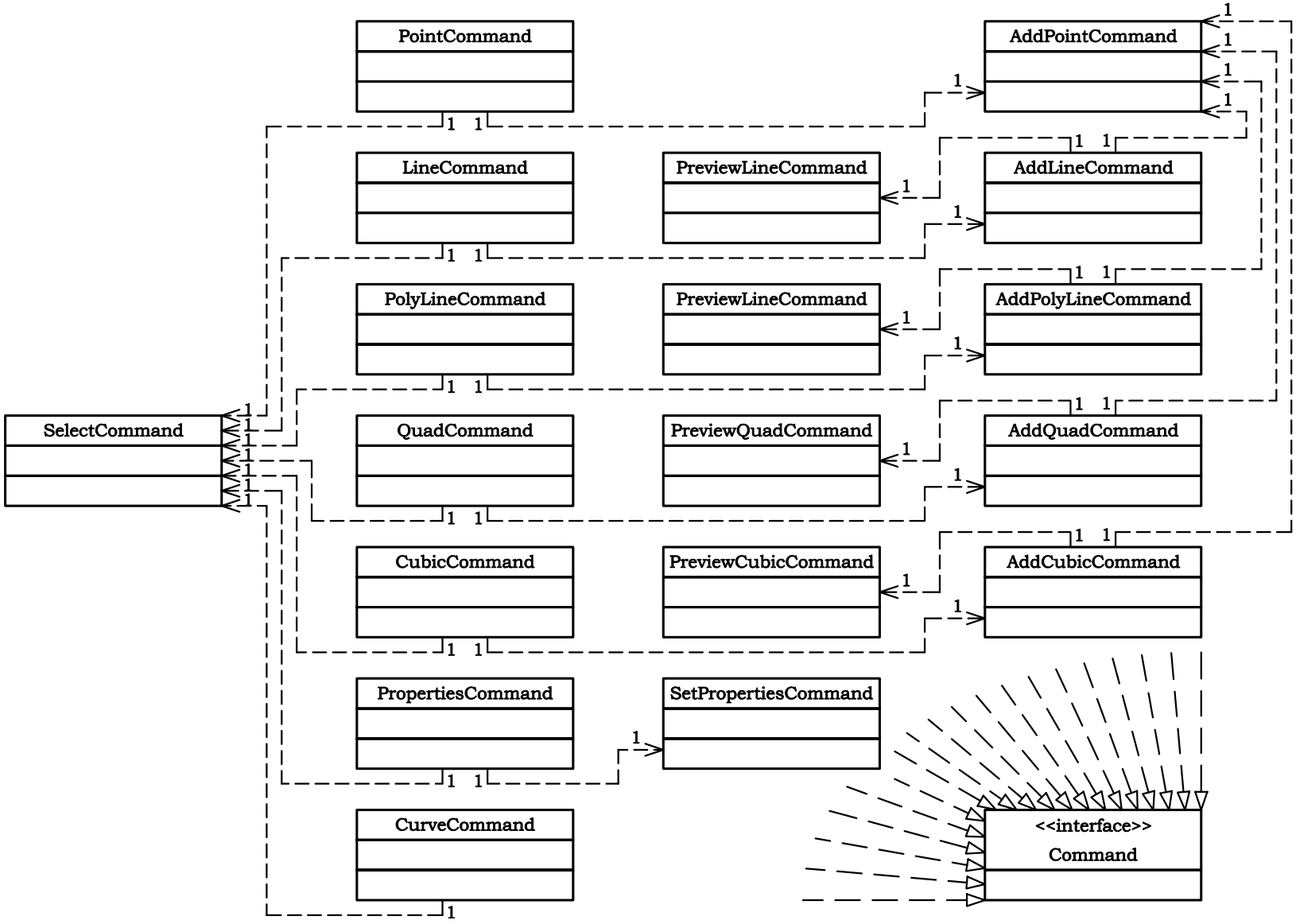


Figura 4.39: Diagrama de herança e instância das classes do subpacote `gui.command`, as linhas que representam a herança foram interrompidas para facilitar a compreensão do diagrama

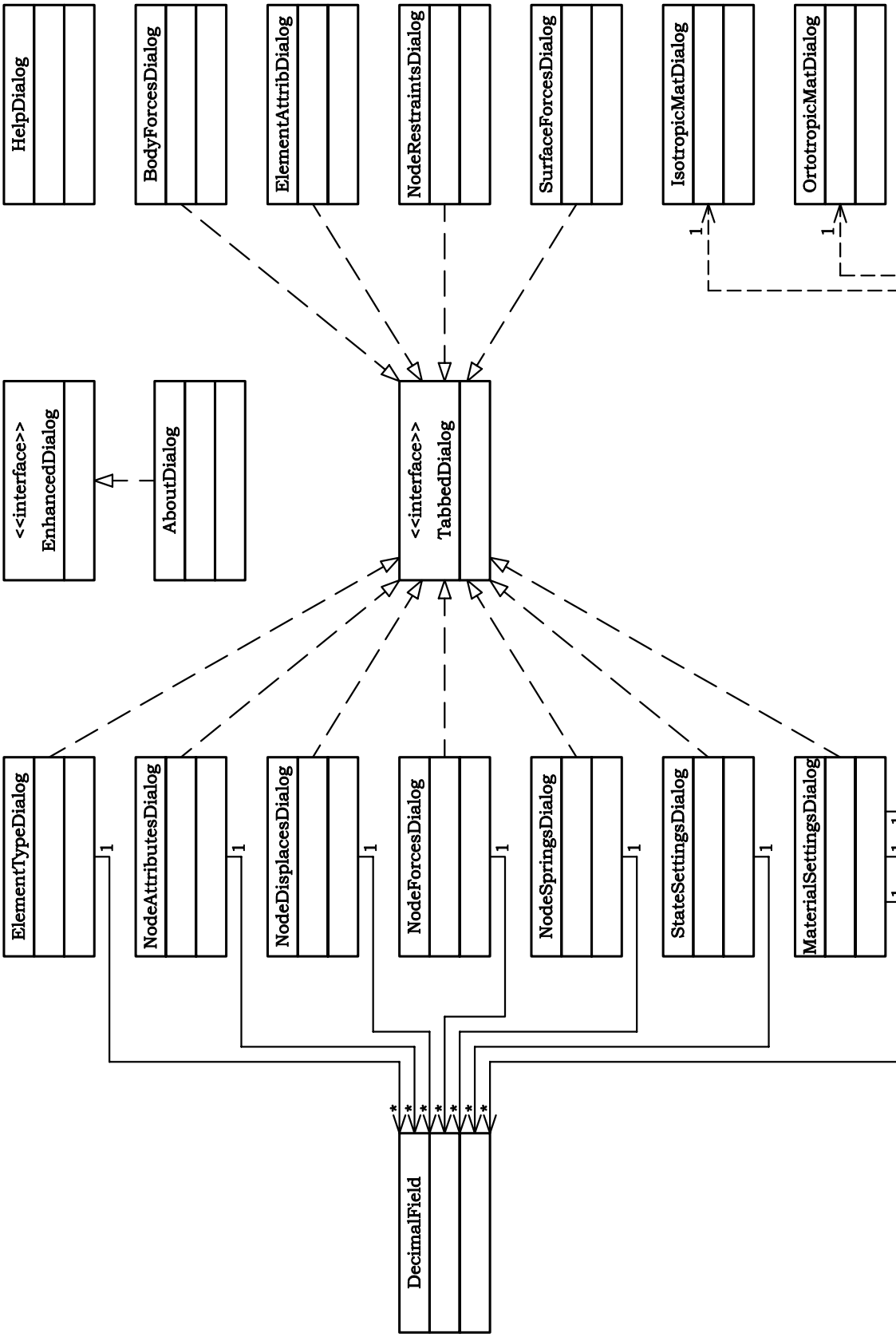


Figura 4.40: Diagrama de herança e instância das classes do subpacote `gui.dialog`

4.8 Integração entre as Classes

Como dito anteriormente, o relacionamento entre as classes criadas para implementação do `MVC` é conseguido com o uso do padrão `Comando`. A figura 4.41, que detalha a figura 4.16 (página 51), exemplifica este relacionamento.

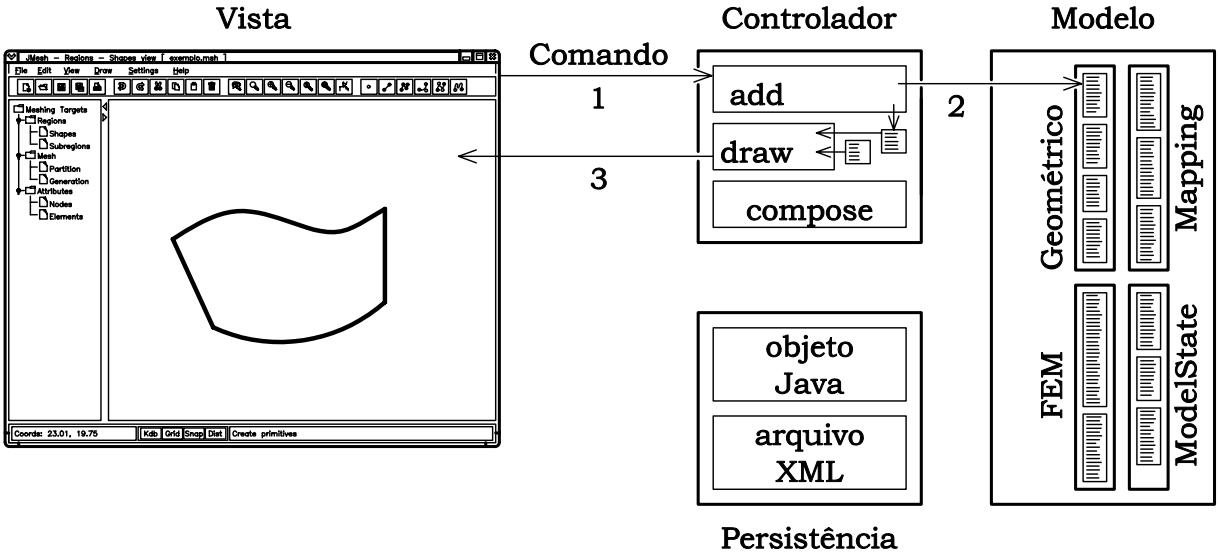


Figura 4.41: Relacionamento entre camadas para adição de uma entidade geométrica

Tendo o usuário requisitado a introdução de uma entidade geométrica ao Modelo, o Controlador correspondente (figura 4.35) é acionado através do Comando pertinente (figura 4.39), recebendo deste a requisição para adicionar a entidade (etapa 1 da figura 4.41).

Com informações relativas à entidade a ser adicionada, o Controlador solicita a adição da mesma ao Modelo (figura 4.23), instancia um objeto de desenho correspondente à entidade geométrica (figura 4.33) e adiciona este em sua lista de desenho adequada (etapa 2 da figura 4.41).

Retomando o controle da operação, o Comando solicita à Vista que atualize sua Área de Desenho. Para tanto, o Controlador corrente é acionado para compor o objeto `Graphics2D` a ser desenhado (etapa 3 da figura 4.41).

A figura 4.42, que detalha a figura 4.17 (página 51), mostra o relacionamento entre as camadas para a atividade *abrir um modelo salvo em arquivo XML*. O processo começa quando a `Interface` recebe a requisição do usuário através de um de seus componentes de interface gráfica. O acionamento deste componente provoca a execução do `Command` adequado para controlar o processo.

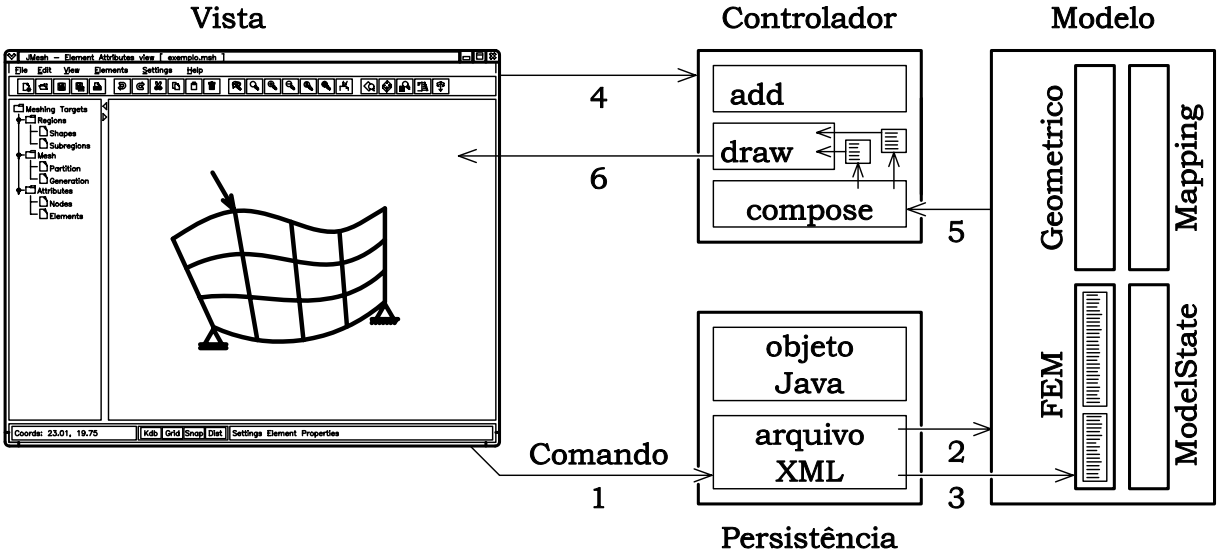


Figura 4.42: Relacionamento entre camadas para abrir um arquivo XML

Na primeira etapa, o `Command` acessa o arquivo XML e cria objetos Java correspondentes aos dados armazenados no arquivo.

A etapa 2 representa a criação de um novo modelo (classe `Model`) que é associado à `Interface` e as configurações (`ModelState`) permanecem inalteradas.

A terceira etapa representa a atribuição do `ParametricModel`, criado na primeira etapa, ao `Model`, criado na segunda etapa. Os demais modelos (GeometricModel e MappingModel) permanecem vazios, uma vez que os mesmos não são persistidos.

Na quarta etapa, um controlador (classe `Controller`) é instanciado e associado à `Interface`. Na quinta etapa, o método `compose()` do `Controller` é executado, preenchendo as listas de desenhos do `Controller` com objetos de desenho representativos do `Model`.

Finalmente, na sexta etapa, é executado o método `draw()` do `Controller`. Este método acessa as listas de desenhos criadas na etapa anterior e compõe o objeto `Graphics2D` a ser apresentado na área de desenho da `Interface`.

A figura 4.43, que detalha a figura 4.18 (página 52), mostra o relacionamento entre as camadas para a atividade *abrir um modelo salvo como objeto Java*. O processo, assim como o anterior, começa quando a `Interface` recebe a requisição do usuário através de um de seus componentes de interface gráfica. O acionamento deste componente provoca a execução do `Command` adequado para controlar o processo.

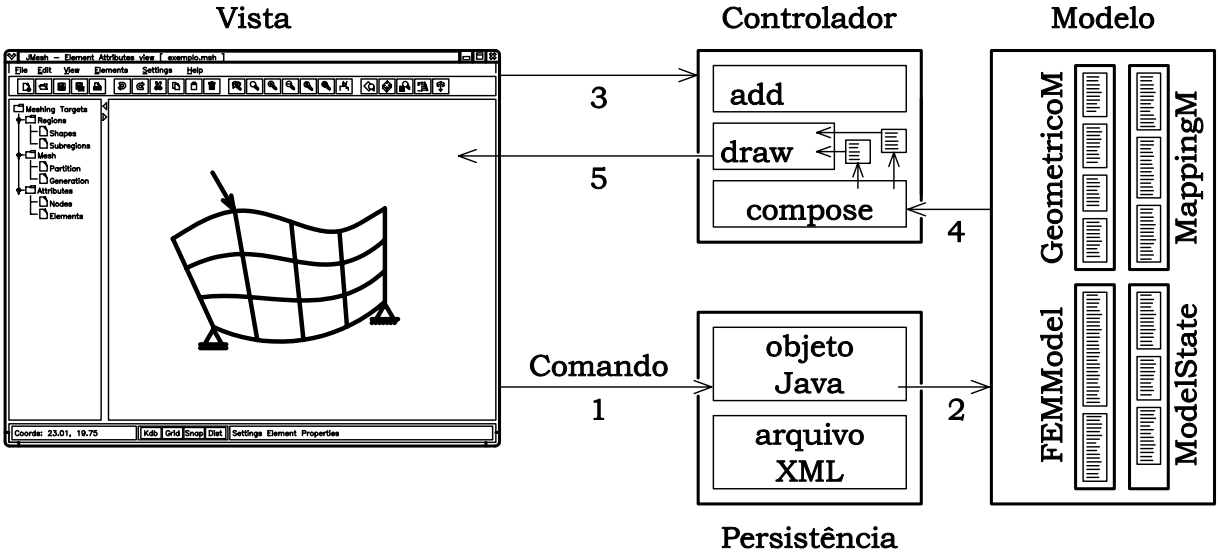


Figura 4.43: Relacionamento entre camadas para abrir um arquivo binário

Na primeira etapa, o `Command` acessa o arquivo persistido e obtém um objeto `Model`.

Na etapa 2, o objeto `Model`, obtido na etapa anterior, é associado à `Interface`. Diferentemente do processo mostrado anteriormente, todo o `Model` é persistido.

Na terceira etapa, um controlador (classe `Controller`) adequado é instanciado e associado à `Interface`. Na quarta etapa, o método `compose()` do `Controller` é executado, preenchendo as listas de desenhos do `Controller` com objetos de desenho representativos do `Model`.

Finalmente, na quinta etapa, é executado o método `draw()` do `Controller`. Este método acessa as listas de desenhos criadas na etapa anterior e compõe o objeto `Graphics2D` a ser apresentado na área de desenho da `Interface`.

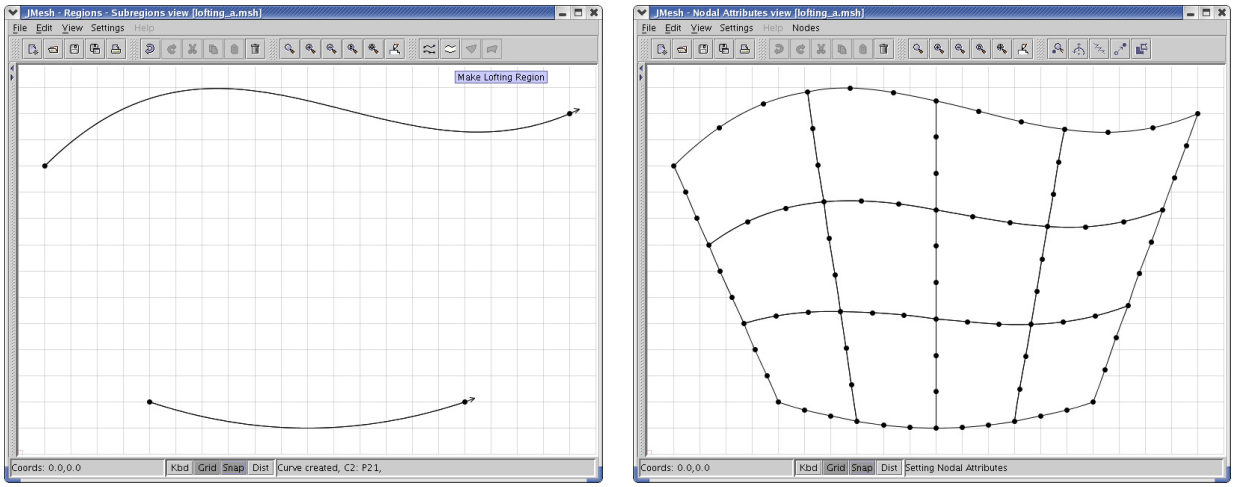
As interações entre as classes do sistema ficam melhor representadas através de diagramas de seqüência. Por uma questão de clareza do texto, os diagramas de seqüência das principais operações da aplicação não são apresentados neste capítulo, mas estão detalhados no Manual de Desenvolvimento do Sistema (Brugiolo & Pitangueira 2004*a*).

Capítulo 5

EXEMPLOS DE GERAÇÃO DE MALHAS

5.1 Mapeamentos "Lofting"

Apesar de sua simplicidade, o mapeamento "lofting" possibilita a geração de diversos tipos de malhas. Na figura 5.1a duas curvas abertas são usadas como contorno, definindo uma região quadrilateral. Após determinar o número de divisões de cada primitiva que compõe as curvas e escolher o tipo de elemento a ser gerado na região definida, obtem-se uma malha inicial (figura 5.1b). Deve-se então especificar os atributos e carregamentos dos nós e elementos para obtenção da malha final, que será persistida em arquivo XML para o programa de análise via Método dos Elementos Finitos.



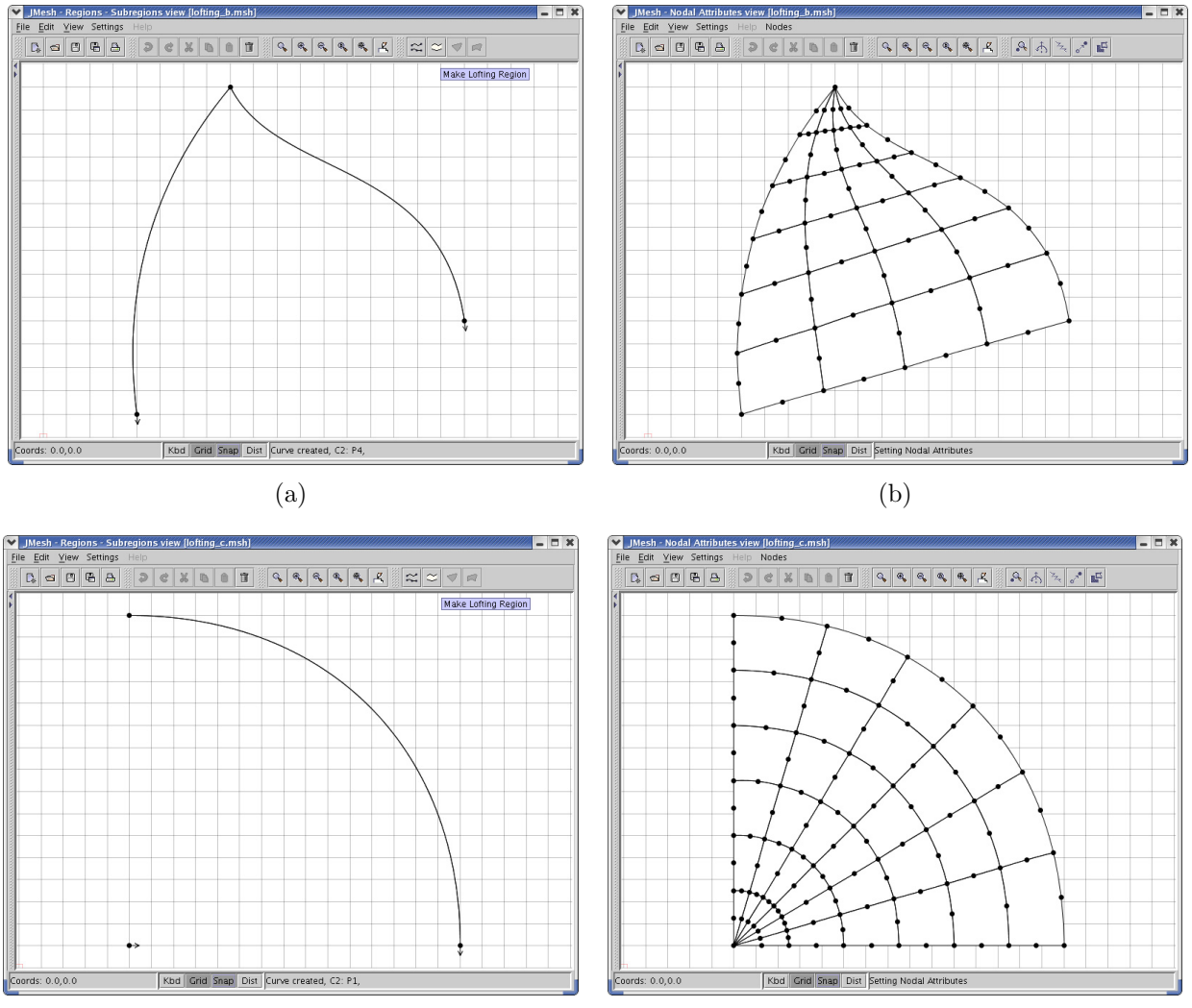
(a) (b)

Figura 5.1: Exemplo de malha gerada por mapeamento "lofting"

Na figura 5.2a, duas curvas abertas, usadas como contorno, apresentam uma extremidade em comum, definindo uma região triangular. Esse procedimento provoca um adensamento da malha em torno do ponto em comum (figura 5.2b). Dessa forma é possível utilizar mapeamentos "lofting" para mapear triangulares.

Na figura 5.2c, uma das curvas usadas como contorno é reduzida a um ponto, definindo uma região triangular. Esse procedimento provoca um adensamento da malha em torno da curva degenerada (figura 5.2d).

A limitação desse processo é a aproximação linear que ocorre entre as curvas ou seja, a região triangular fica definida pelas duas curvas e pelas retas que unem suas extremidades.



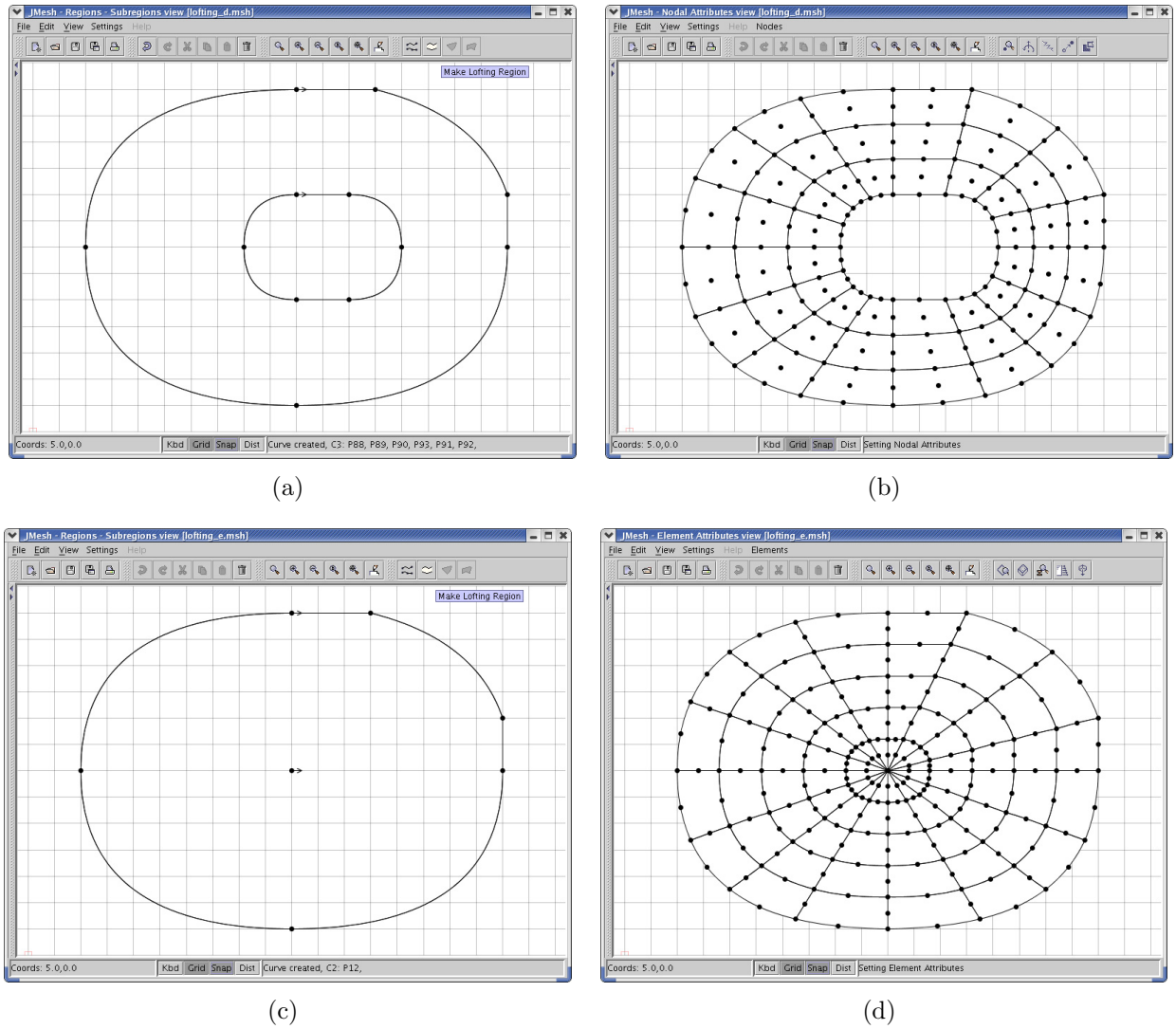
(a) (b)

(c) (d)

Figura 5.2: Exemplos de malhas geradas por mapeamentos "lofting"

Na figura 5.3a duas curvas fechadas e concêntricas são usadas como contorno, a curva externa define os limites da região e a interna delimita o furo (figura 5.3b). O deslocamento do furo em relação ao centro da região pode provocar distorções na malha gerada, e curvas geradas em sentidos contrários provocam inconsistência na malha. Um ângulo de defazagem obtuso entre os pontos correspondentes em cada curva também gera inconsistência na malha.

Na figura 5.3c é usada uma curva fechada e uma curva degenerada em um ponto no interior da primeira. A curva externa define os limites da região e a segunda define o centro da malha radial gerada (figura 5.3d). Mais uma vez ocorre adensamento da malha em torno da curva degenerada.

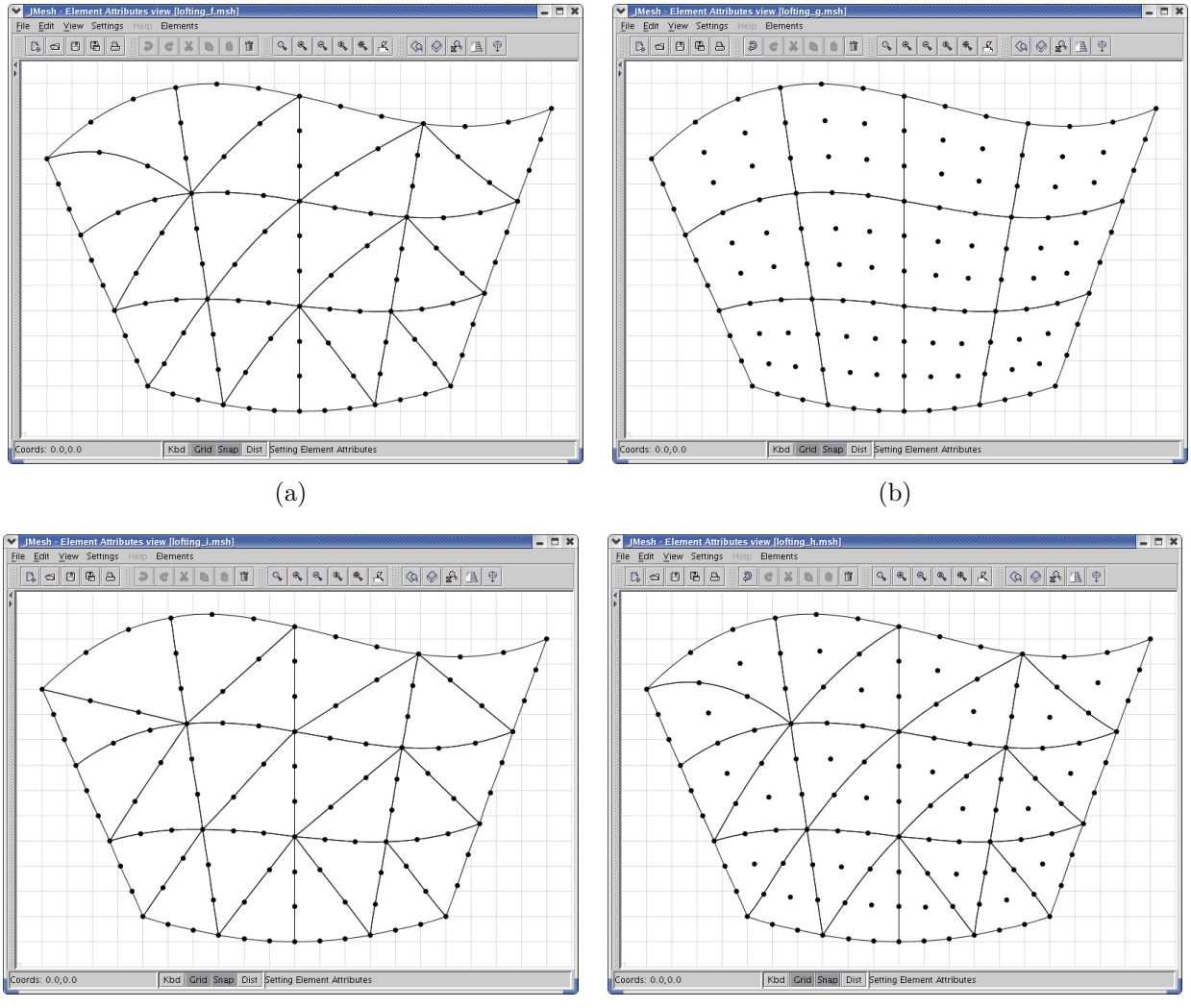


(a) (b)

(c) (d)

Figura 5.3: Exemplos de malhas geradas por mapeamentos "lofting"

No exemplo da figura 5.1b foi escolhido um elemento *serendiptico quadrilateral* de 12 nós. Diversos tipos de elementos poderiam ter sido utilizados, como os *lagrangeanos* (figura 5.4b e figura 5.4d) e os *triangulares* (figura 5.4a, figura 5.4c e figura 5.4d). O projetor "lofting" cria uma partição natural da região, composta por elementos quadrilaterais (figura 5.4b), que podem ser subdivididos em suas diagonais para formarem elementos triângulares (figura 5.4c). Os elementos triangulares são originados pela divisão dos elementos quadrilaterais em suas menores diagonais para que sejam gerados elementos menos distorcidos. Pode-se ainda exigir que os nós que formam a diagonal menor sejam alinhados (figura 5.4c) para controlar possíveis distorções.



(a) (b)

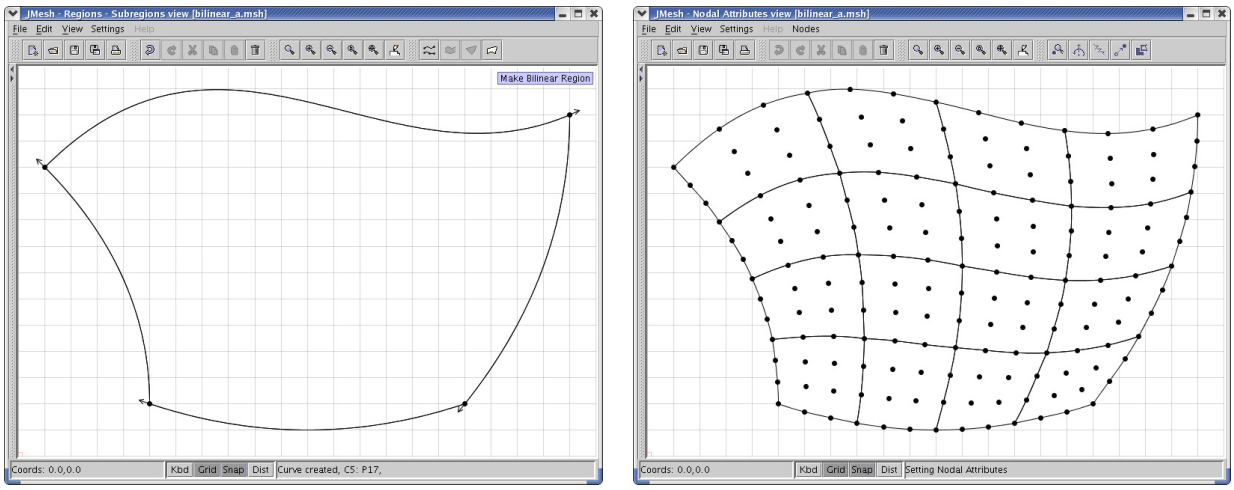
(c) (d)

Figura 5.4: Exemplos de malhas geradas por mapeamentos "lofting"

5.2 Mapeamentos Bilineares

O mapeamento transfinito bilinear é capaz de gerar qualquer uma das malhas mostradras na seção anterior (5.1 Mapeamentos “Lofting”) e define com igual precisão qualquer região da seção seguinte (5.3 Mapeamentos Trilineares). Porém, o uso de mapeamento bilinear em regiões triangulares gera um adensamento da malha em torno da curva que se degenera em um ponto, o que pode ser indesejado.

Na figura 5.5a, quatro curvas em sequência são usadas como contorno, definindo uma região quadrilateral. O mapeamento transfinito bilinear é o mais adequado em regiões com contornos curvos nos quatro lados. Após determinar o número de divisões de cada primitiva que compõem as curvas e escolher o tipo de elemento a ser gerado na região definida, obtem-se uma malha inicial (figura 5.5b). Deve-se então especificar os atributos e carregamentos dos nós e elementos para obtenção da malha final, que será persistida em arquivo XML para o programa de análise via Método dos Elementos Finitos.



(a) (b)

Figura 5.5: Exemplo de malha gerada por mapeamento bilinear

Na figura 5.6a, uma das curvas que delimitam a região é linear e outra foi reduzida a um ponto, definindo uma região triangular com um lado reto. Esse procedimento provoca um adensamento da malha em torno da curva reduzida a um ponto (figura 5.6b).

Na figura 5.6c, duas das curvas que delimitam a região são lineares e outra foi reduzida a um ponto, definindo uma região triangular com dois lados retos. Nesse procedimento também

ocorre adensamento da malha em torno da curva reduzida a um ponto (figura 5.6d).

Nos dois exemplos (figura 5.6b e figura 5.6d) a curva reduzida a um ponto poderia estar situada entre duas curvas quaisquer, permitindo deslocar o adensamento da malha para o ponto mais conveniente.

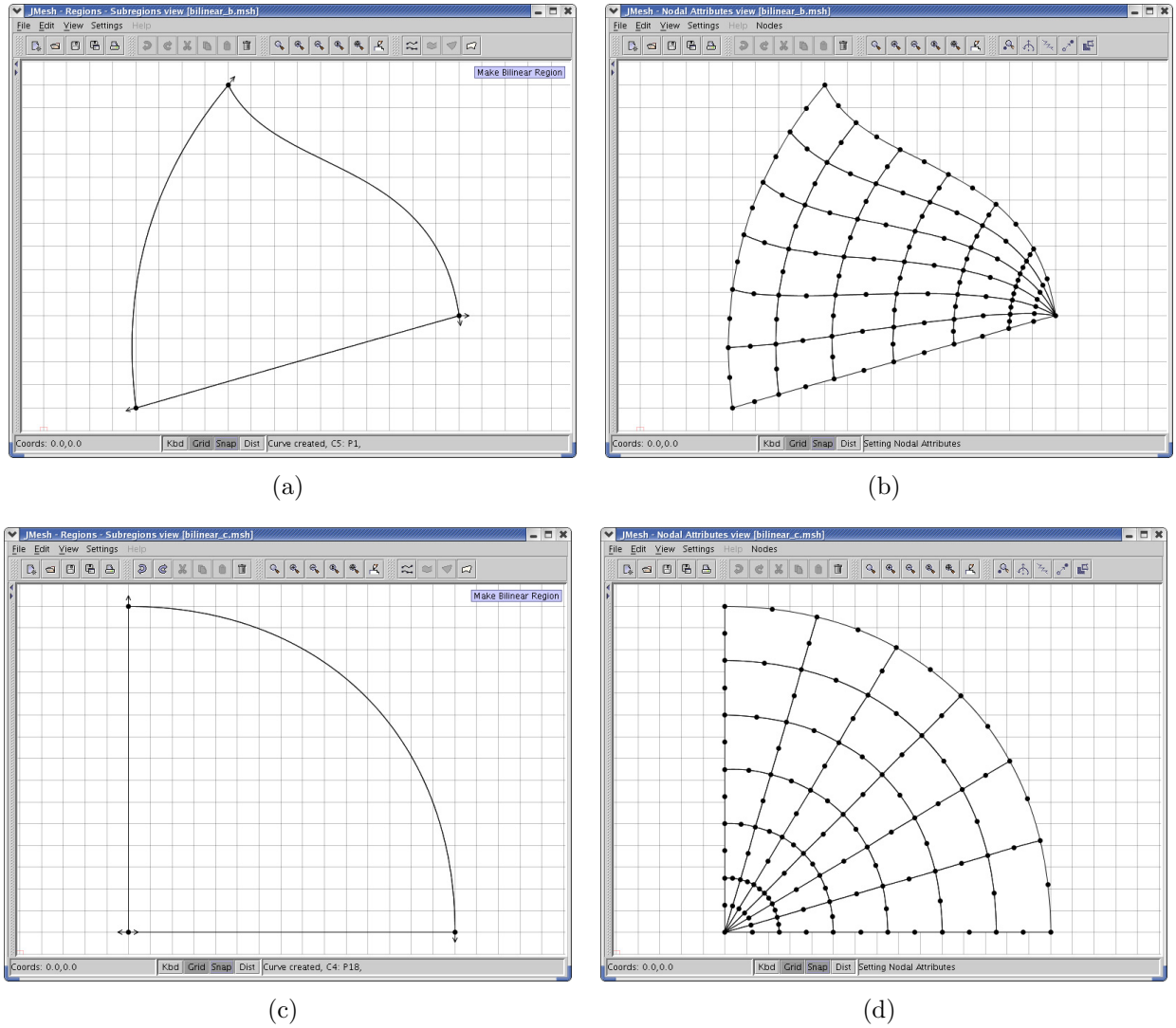
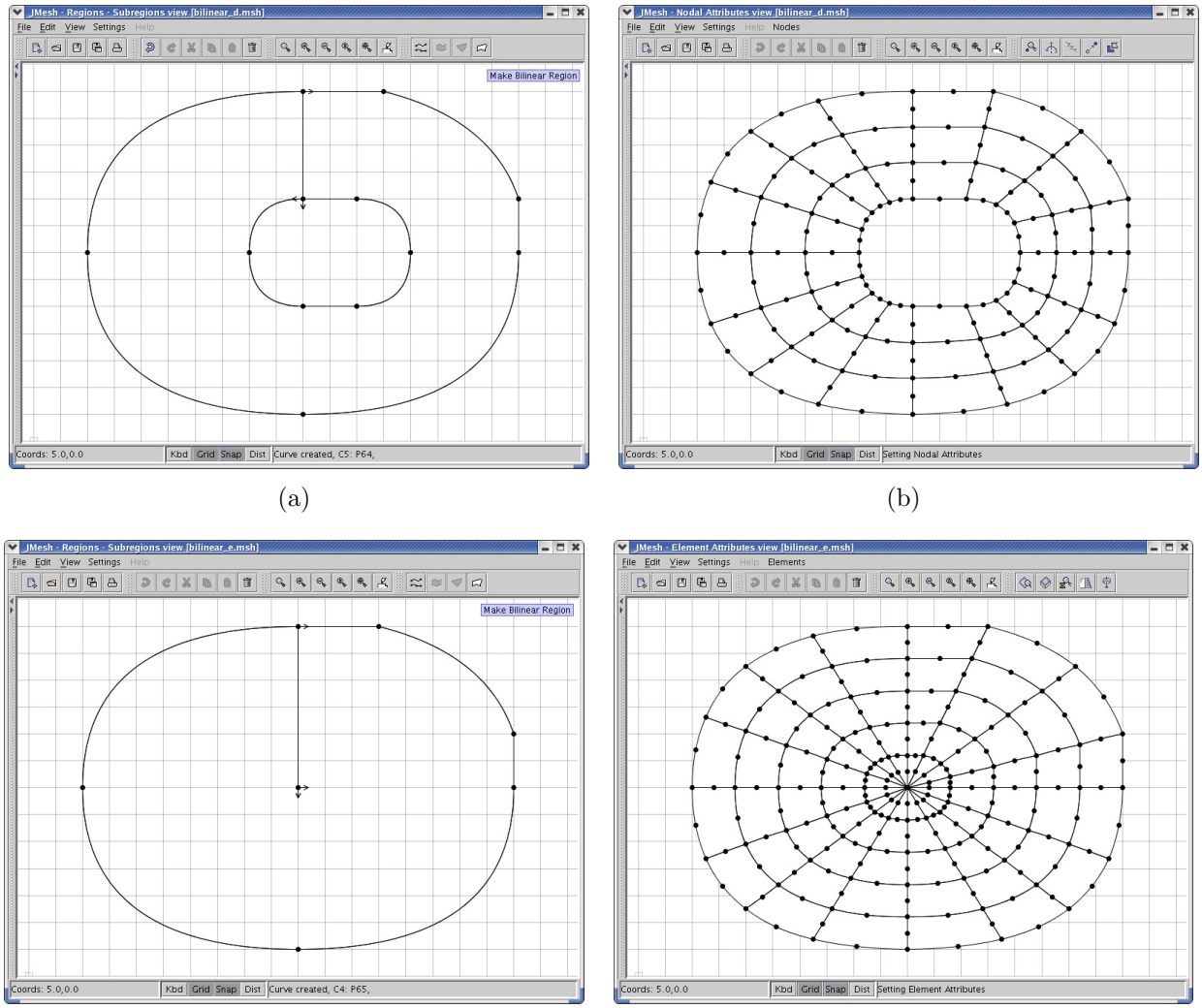


Figura 5.6: Exemplos de malhas geradas por mapeamentos bilineares

Na figura 5.7a, duas das curvas usadas como contorno são fechadas e concêntricas, a curva externa define os limites da região e a interna delimita o furo (figura 5.7b). As demais curvas ligam as extremidades das duas primeiras. O deslocamento do furo em relação ao centro da região pode provocar distorções na malha gerada, e as duas curvas fechadas devem ser geradas em sentidos contrários para que não ocorra inconsistência na malha. Um ângulo de defazagem

obtuso entre os pontos correspondentes em cada curva também gera inconsistência na malha.

Na figura 5.7c, é usada uma curva fechada e uma curva degenerada em um ponto no interior da primeira. A curva externa define os limites da região e a segunda define o centro da malha radial gerada (figura 5.7d). As demais curvas ligam as extremidades das duas primeiras. Mais uma vez ocorre adensamento da malha em torno da curva degenerada.



(a) (b)

(c) (d)

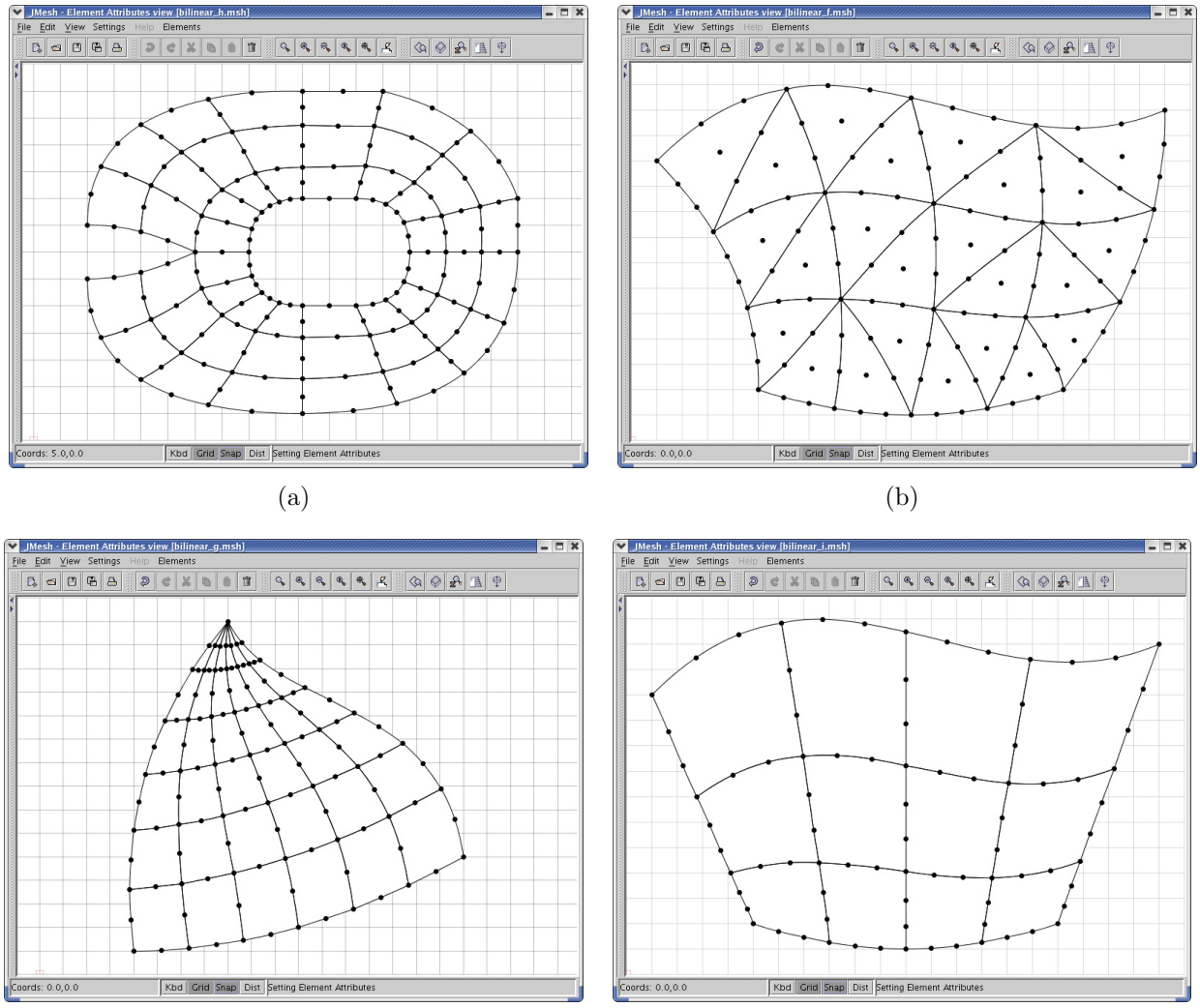
Figura 5.7: Exemplos de malhas geradas por mapeamentos bilineares

Nos dois exemplos anteriores (figura 5.7b e figura 5.7d) as curvas que fazem a ligação entre as extremidades das curvas internas e externas permitem um maior controle sobre a malha gerada (figura 5.8a).

Na figura 5.8b é apresentada uma malha de elementos *lagrangeanos triangulares* de 10 nós. Esses elementos foram gerados pela divisão dos elementos apresentados na figura 5.5b.

Na figura 5.8c uma das curvas que delimitam a região foi reduzida a um ponto, definindo uma região triangular com três lados curvos. Mais uma vez ocorre adensamento da malha em torno da curva reduzida a um ponto.

Na figura 5.8d, são usadas duas curvas lineares não consecutivas para definir uma região idêntica a apresentada na figura 5.1a.



(a) (b)

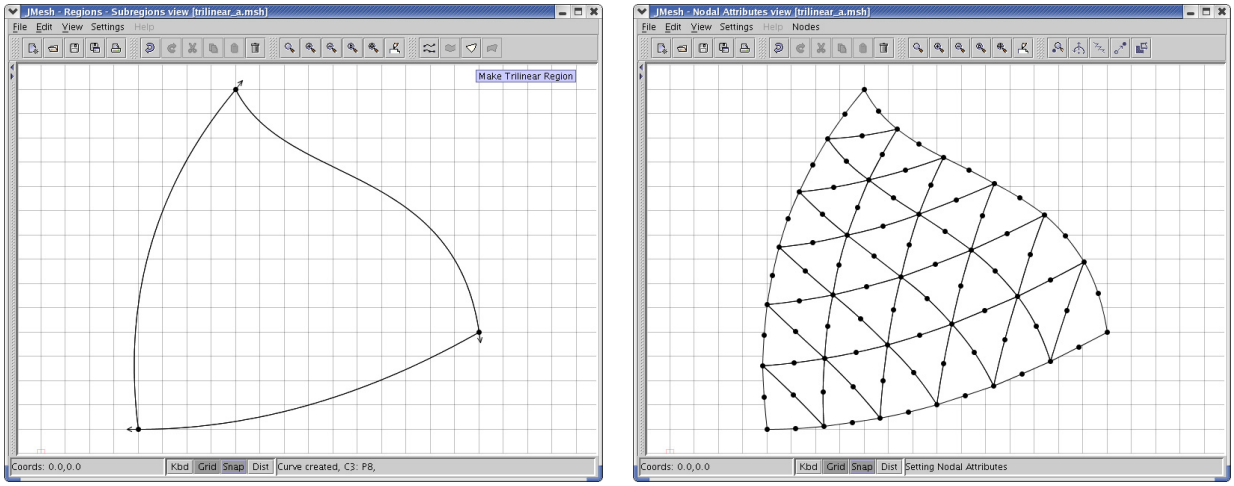
(c) (d)

Figura 5.8: Exemplos de malhas geradas por mapeamentos bilineares

5.3 Mapeamentos Trilineares

Quando os mapeamentos "lofting" e bilinear são usados para mapear regiões triangulares, a malha resultante fica adensada em torno da curva que se degenera em um ponto. Quando desejado, esse adensamento pode ser evitado usando o mapeamento transfinito trilinear para gerar malhas em regiões limitadas por três curvas (figura 5.9a). O Posicionamento dos pontos de controle de cada primitiva provocam uma variação *gradual* na densidade da malha gerada.

Na figura 5.9a três curvas em sequência são usadas como contorno, definindo uma região triangular. Após determinar o número de divisões de cada primitiva que compõe as curvas e escolher o tipo de elemento a ser gerado na região definida, obtem-se uma malha inicial (figura 5.9b). Observa-se que a malha é uniforme, não apresentando adensamento algum. Deve-se então especificar os atributos e carregamentos dos nós e elementos para obtenção da malha final, que será persistida em arquivo XML para o programa de análise via Método dos Elementos Finitos.



(a) (b)

Figura 5.9: Exemplo de malha gerada por mapeamento trilinear

Para gerar a malha da figura 5.10a, foram utilizadas três curvas para definir o contorno da região, uma delas é linear, definindo uma região triangular com um lado reto. Para gerar a malha da figura 5.10b, foram utilizadas três curvas para definir o contorno da região, duas delas são lineares, definindo uma região triangular com dois lados retos.

Na figura 5.10c, três curvas definem uma região triangular. O posicionamento dos pontos de controle das primitivas que compõem o lado direito e esquerdo da região promoveu uma redução da densidade da malha em torno do vértices superior da região triangular, e um aumento gradual na densidade da malha a medida que se aproxima do lado inferior da região (figura 5.10d).

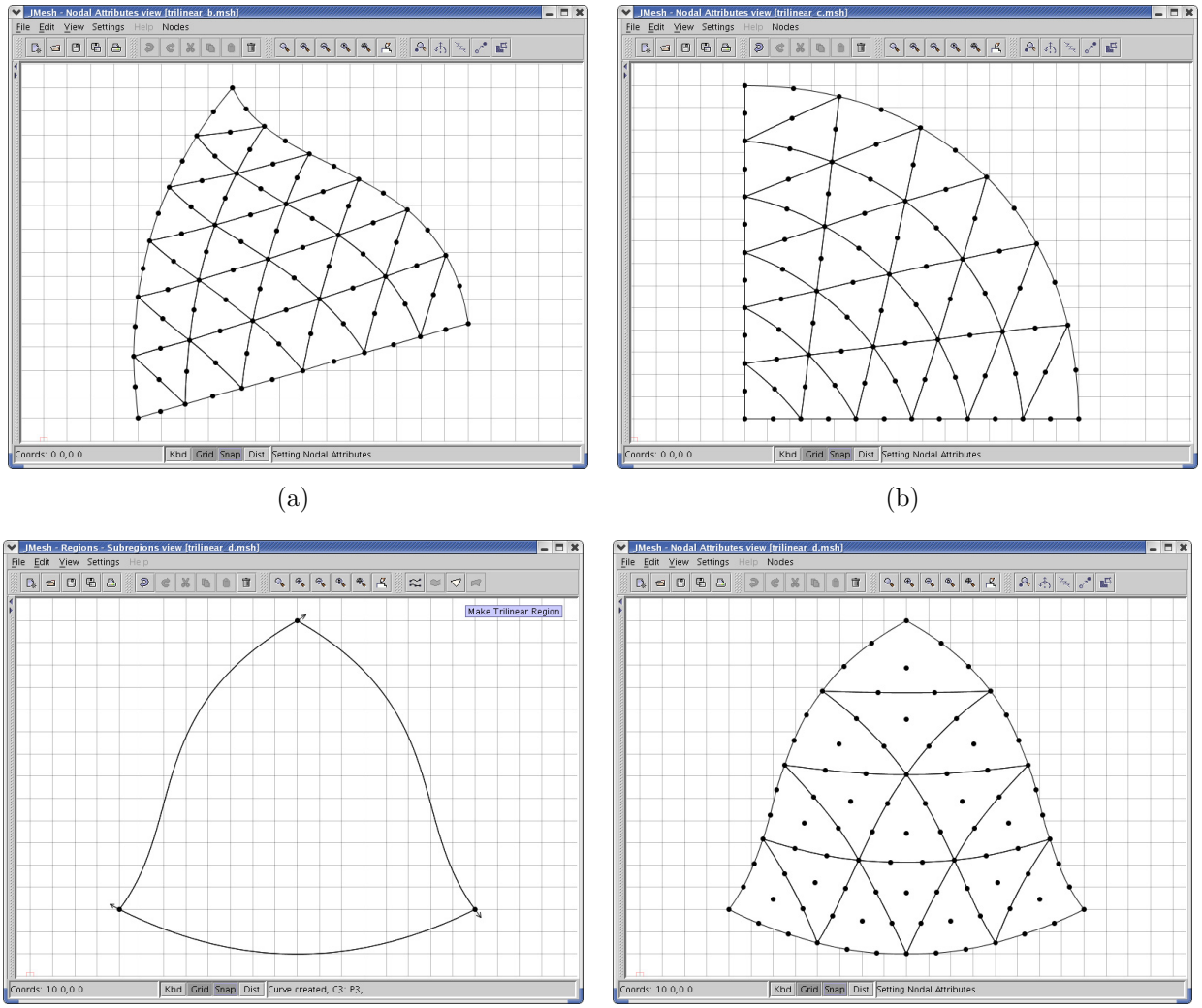


Figura 5.10: Exemplos de malhas geradas por mapeamentos trilineares

Nas figuras 5.11a e 5.11c, apesar de apresentarem contornos compostos por três lados retos, foram utilizadas primitivas cúbicas para definir suas respectivas curvas e regiões. O uso de primitivas cúbicas possibilitou que uma variação gradual na densidade da malha fosse obtida

através do delocamento dos pontos de controle de cada primitiva. Na figura 5.11b observa-se um adensamento da malha no centro da região, enquanto na figura 5.11d observa-se um adensamento da malha na periferia da região.

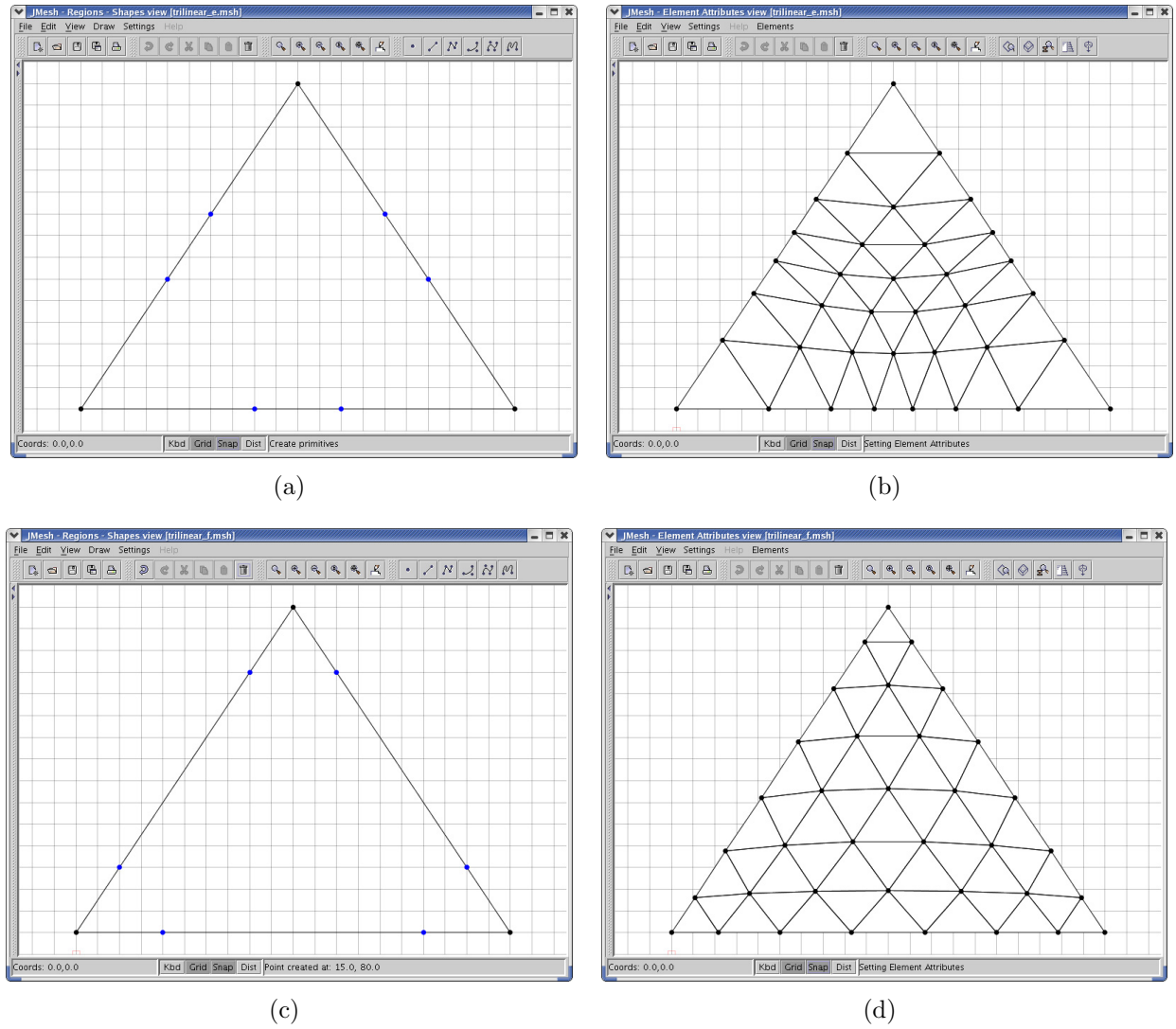


Figura 5.11: Exemplos de malhas geradas por mapeamentos trilineares

Primitivas quadráticas também podem ser utilizadas para promover variação *gradual* na densidade da malha.

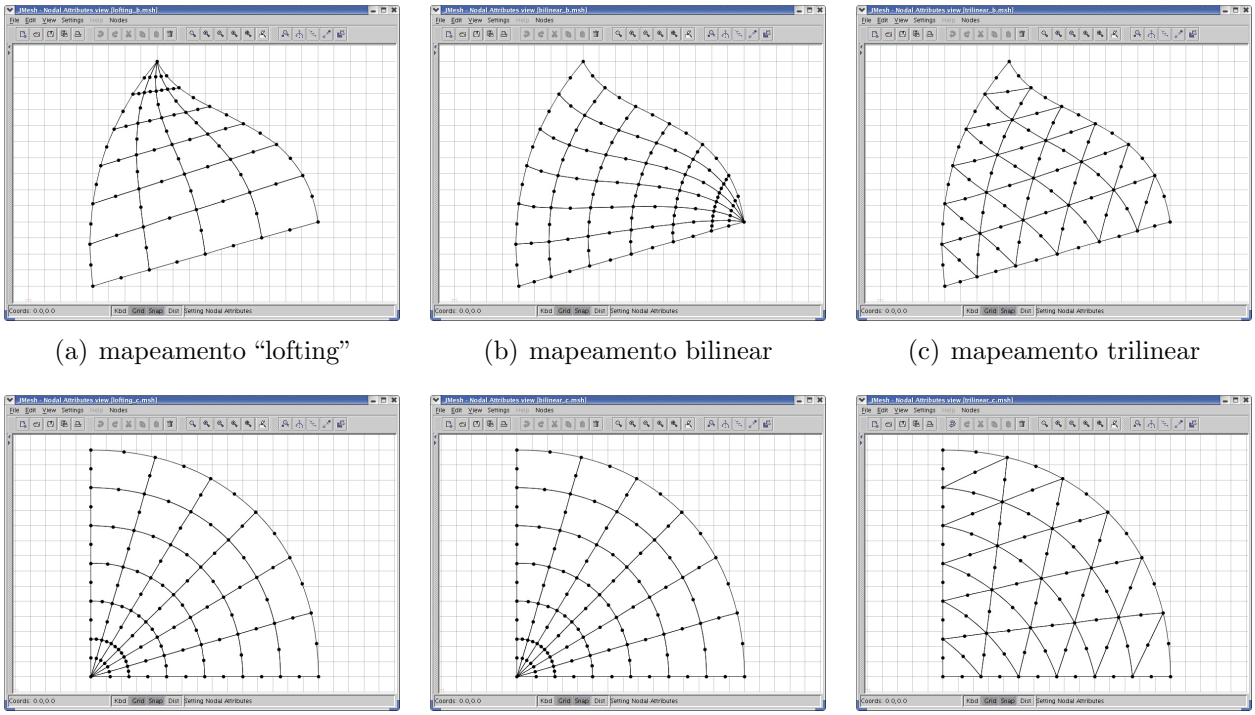
5.4 Comparações

O mapeamento "lofting" é capaz de definir o contorno de regiões triangulares, desde que um ou dois dos lados dessas regiões sejam retos ou aproximações lineares. Sempre ocorrerá

adensamento da malha no ponto em comum entre as duas curvas (figura 5.12a) ou na curva reduzida a um ponto (figura 5.12d).

O mapeamento bilinear é capaz de descrever com precisão o contorno de regiões triangulares. Para isso é necessário reduzir uma das curvas do contorno a um ponto em torno do qual ocorre aumento na densidade da malha gerada (figura 5.12b e 5.12e).

O mapeamento trilinear é capaz de discrever com precisão o contorno de regiões triangulares e impede a ocorrência de variações na densidade da malha (figura 5.12c e 5.12f).



(a) mapeamento "lofting" (b) mapeamento bilinear (c) mapeamento trilinear

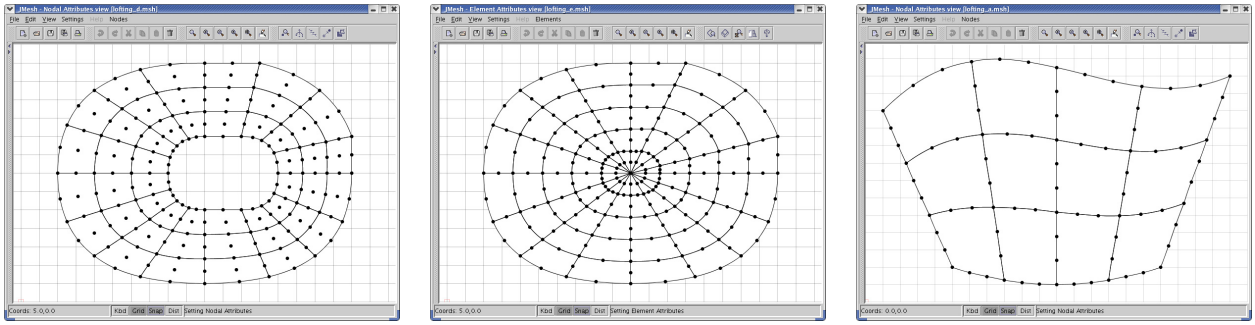
(d) mapeamento "lofting" (e) mapeamento bilinear (f) mapeamento trilinear

Figura 5.12: Exemplos de malhas triangulares

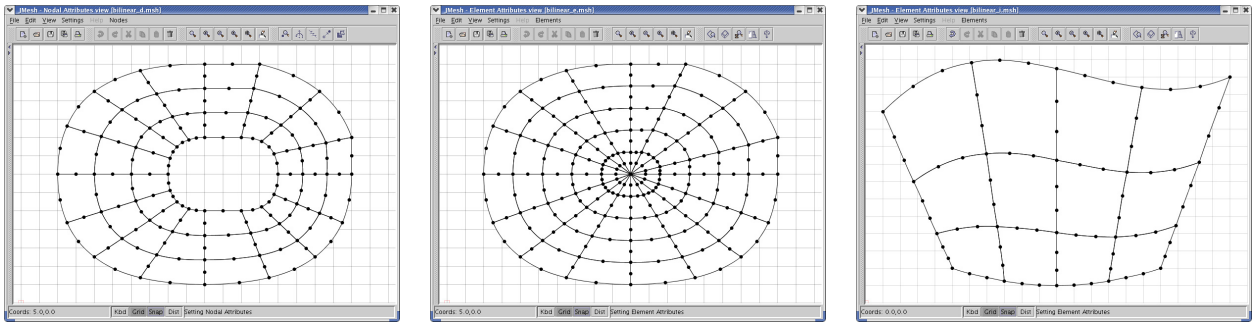
As figuras 5.12a e 5.12b apresentam malhas diferentes para uma mesma região. Apesar disso, com o mapeamento bilinear, utilizado para gerar a malha da figura 5.12b, poderia ter sido obtida uma malha idêntica a apresentada na figura 5.12a. Para isso, bastaria situar a curva reduzida a um ponto no vértice superior da região triangular.

Utilizando o mapeamento bilinear é possível obter qualquer malha gerada com o mapeamento "lofting". Para isso basta usar as mesmas curvas usadas no mapeamento "lofting" e adicionar duas curvas lineares ligando as extremidades das outras duas. Na figura 5.13 são

apresentadas algumas malhas obtidas com mapeamentos “lofting” (figuras 5.13a, 5.13b e5.13c) e bilineares (figuras 5.13c, 5.13d e5.13e).



(a) mapeamento “lofting” (b) mapeamento “lofting” (c) mapeamento “lofting”



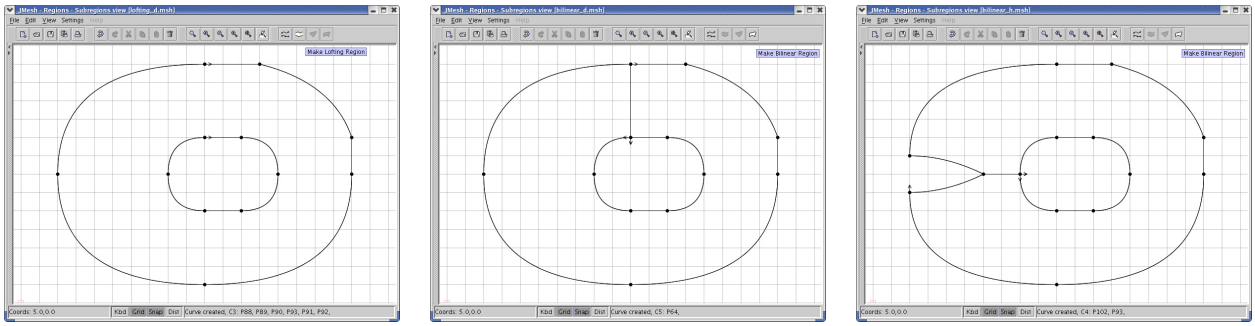
(d) mapeamento bilinear (e) mapeamento bilinear (f) mapeamento bilinear

Figura 5.13: Exemplos de malhas em regiões idênticas

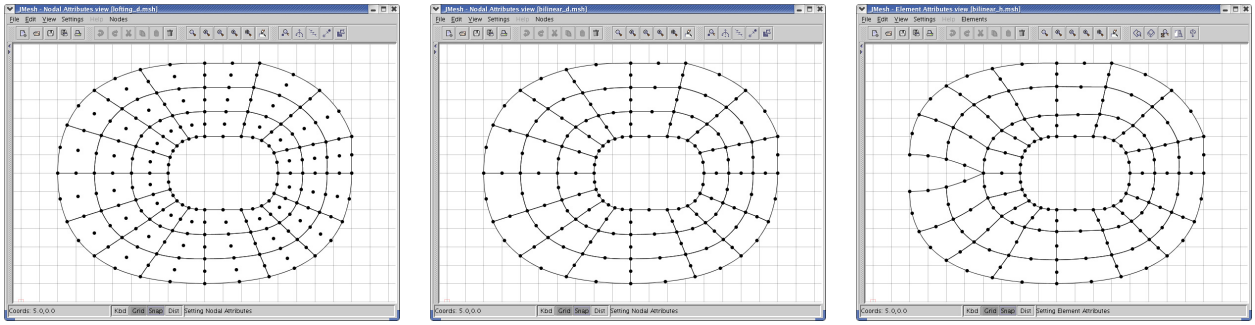
Observar-se que as figuras 5.13c e 5.13f apresentam malhas diferentes para uma mesma região. Isso ocorre porque na malha da figura 5.13f as primitivas que compõem as curvas da direita e da esquerda não são lineares. São primitivas quadráticas com seus pontos de controles alinhados com suas extremidades. Utilizando primitivas quadráticas nas curvas da direita e da esquerda foi possível provocar uma variação gradual da densidade da malha (figuras 5.14f) deslocando-se os pontos de controle. Esse procedimento pode ser usado para reduzir distorções nos elementos de algumas malhas e não pode ser executado com o mapeamento “lofting”. As malhas das figuras 5.13a e 5.13d diferem apenas pela tipo de elemento usado na geração da malha.

A figura 5.14 apresenta outras comparações entre os mapeamentos “lofting”e bilinear. Obser-va-se na figura 5.14c que as curvas que ligam a extremidades das curvas interna e externa, presentes apenas no mapeamento bilinear, possibilitam um controle mais efetivo sobre

a malha gerada, permitindo a inclusão de um entalhe (figura 5.14f).



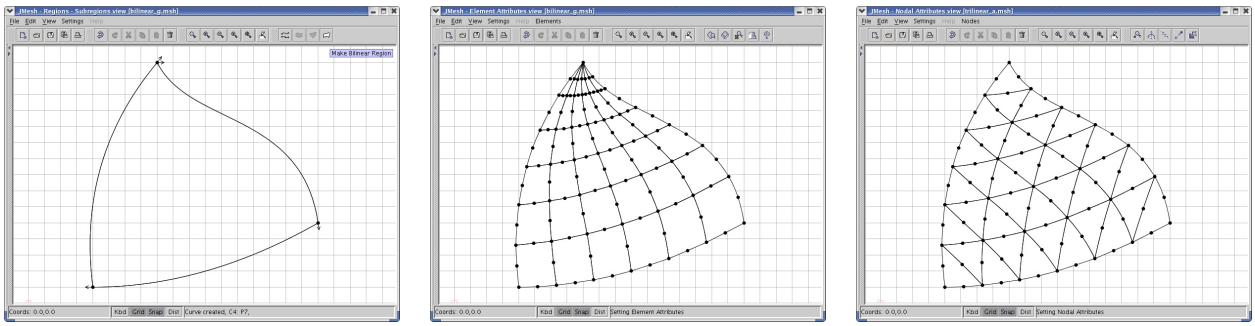
(a) mapeamento “lofting” (b) mapeamento bilinear (c) mapeamento bilinear



(d) mapeamento “lofting” (e) mapeamento bilinear (f) mapeamento bilinear

Figura 5.14: Mapeamento bilinear x “lofting”

Na figura 5.15 o mapeamento bilinear define com precisão a região delimitada por três curvas (figura 5.15a). Mais uma vez ocorre adensamento da malha em torno da curva reduzida a um ponto (figura 5.15b), o mesmo não ocorre com o mapeamento trilinear (figura 5.15c).

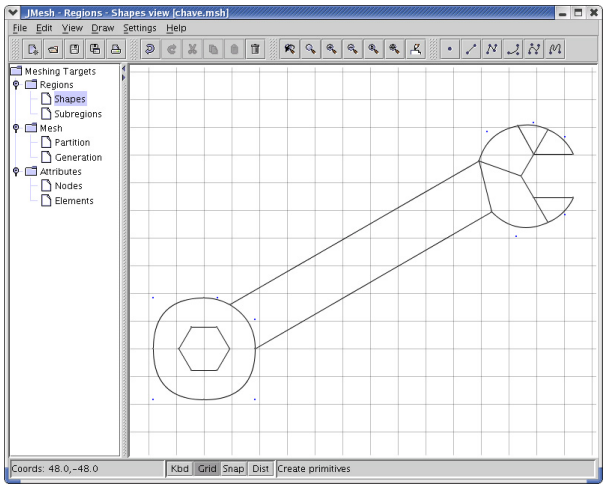


(a) mapeamento bilinear (b) mapeamento bilinear (c) mapeamento trilinear

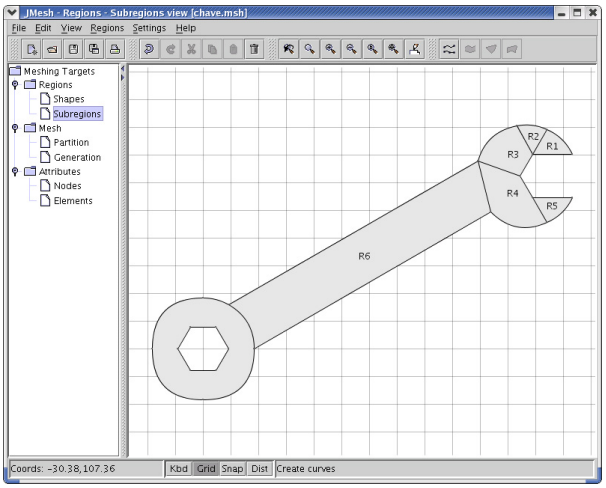
Figura 5.15: Mapeamento bilinear x trilinear

5.5 Combinação dos Mapeamentos

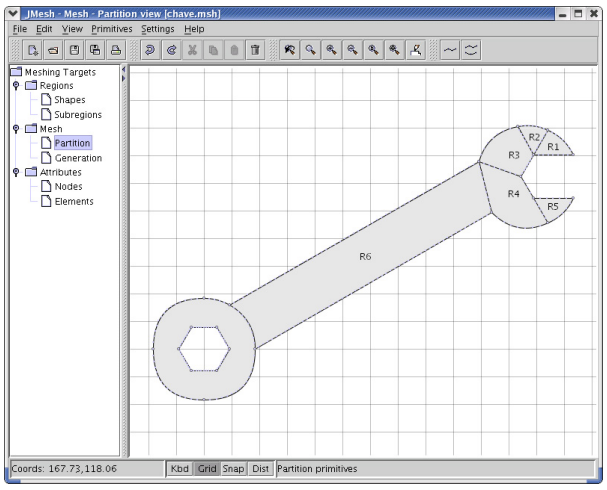
Esta seção apresenta, como exemplo, uma combinação dos mapeamentos "lofting", bilinear e trilinear. A figura 5.16a mostra a definição da geometria do modelo. Observa-se na figura 5.16b a sub-divisão do modelo em diversas sub-regiões com o objetivo que propiciar um maior controle sobre a malha final. Após a divisão das primitivas (figura 5.16c) e determinação do tipo de elemento (serendíptico triangular de seis nós) a ser utilizado as malhas foram geradas (figura 5.16d). Foram utilizados mapeamentos trilineares nas sub-regiões *R1*, *R2* e *R5*, mapeamentos bilineares nas sub-regiões *R3*, *R4* e *R6*, e mapeamento "lofting" na última sub-região.



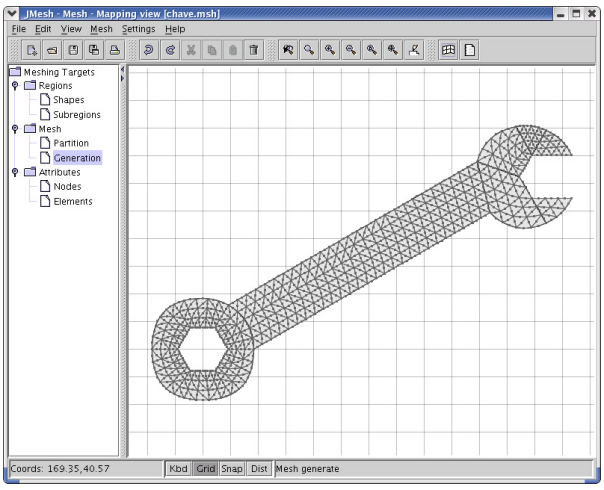
(a) definição da geometria



(b) criação das sub-regiões



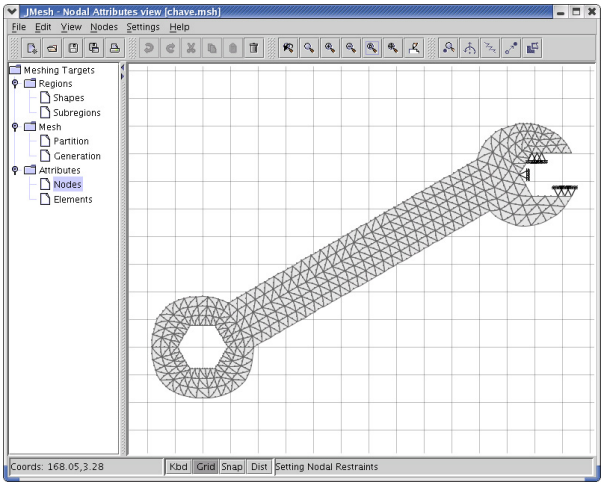
(c) divisão das primitivas



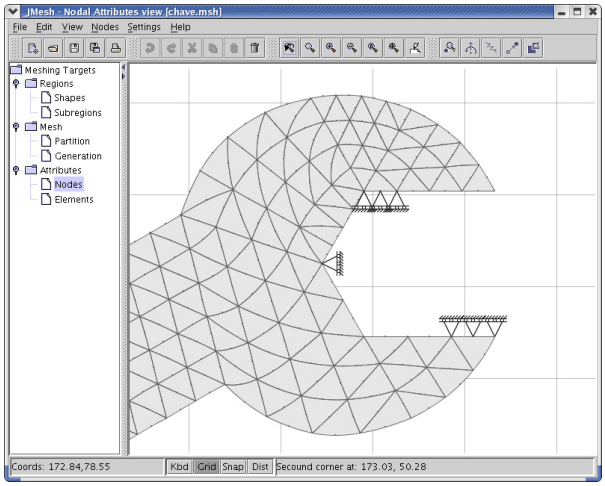
(d) geração das malhas

Figura 5.16: Exemplo com combinação de mapeamentos

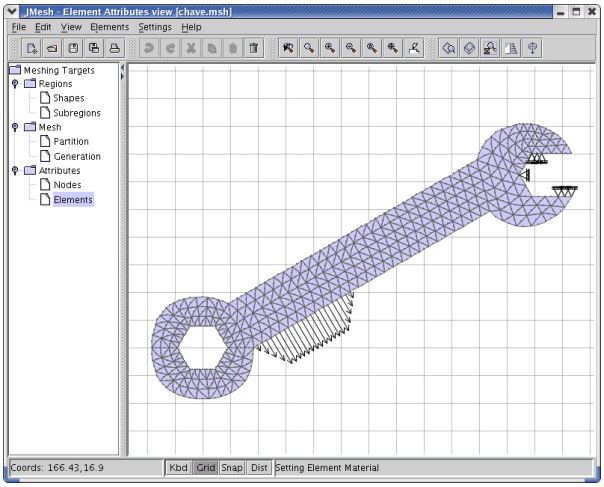
A figura 5.17a apresenta as condições de apoio do modelo (ver detalhe na figura 5.17b). Este exemplo tem o objetivo de ilustrar a combinação de mapeamentos e não a modelagem precisa de um problema real, por isso as condições de apoio não refletem com fidelidade as condições reais, melhor representadas por um conjunto de apoios elásticos com rigidez de mola que refletisse a rigidez da cabeça sextavada de um parafuso. A figura 5.17c apresenta as condições de carregamento do modelo, a modificação da cor dos elementos indica que foi realizado o ajuste do material, representado por esta cor. Para gerar a malha final são ajustadas algumas propriedades dos elementos que não possuem representação como a espessura e o tipo de análise associados a cada elemento.



(a) atributos nodais



(b) detalhe dos apoios



(c) atributos dos elementos

Figura 5.17: Exemplo com combinação de mapeamentos

5.6 Outros Exemplos

As figuras 5.18 e 5.19 mostram outras malhas geradas pela aplicação.

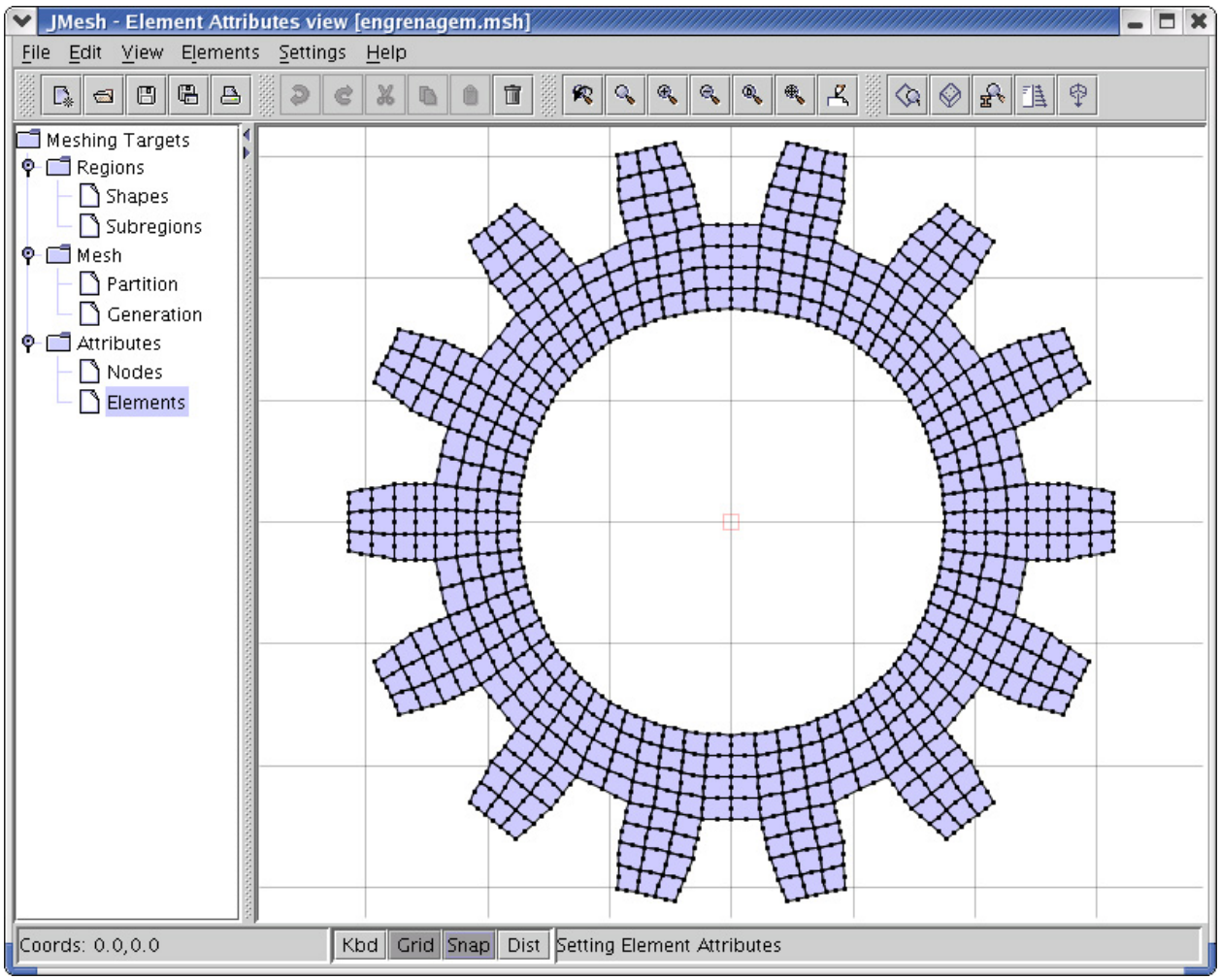


Figura 5.18: Engrenagem, símbolo da engenharia

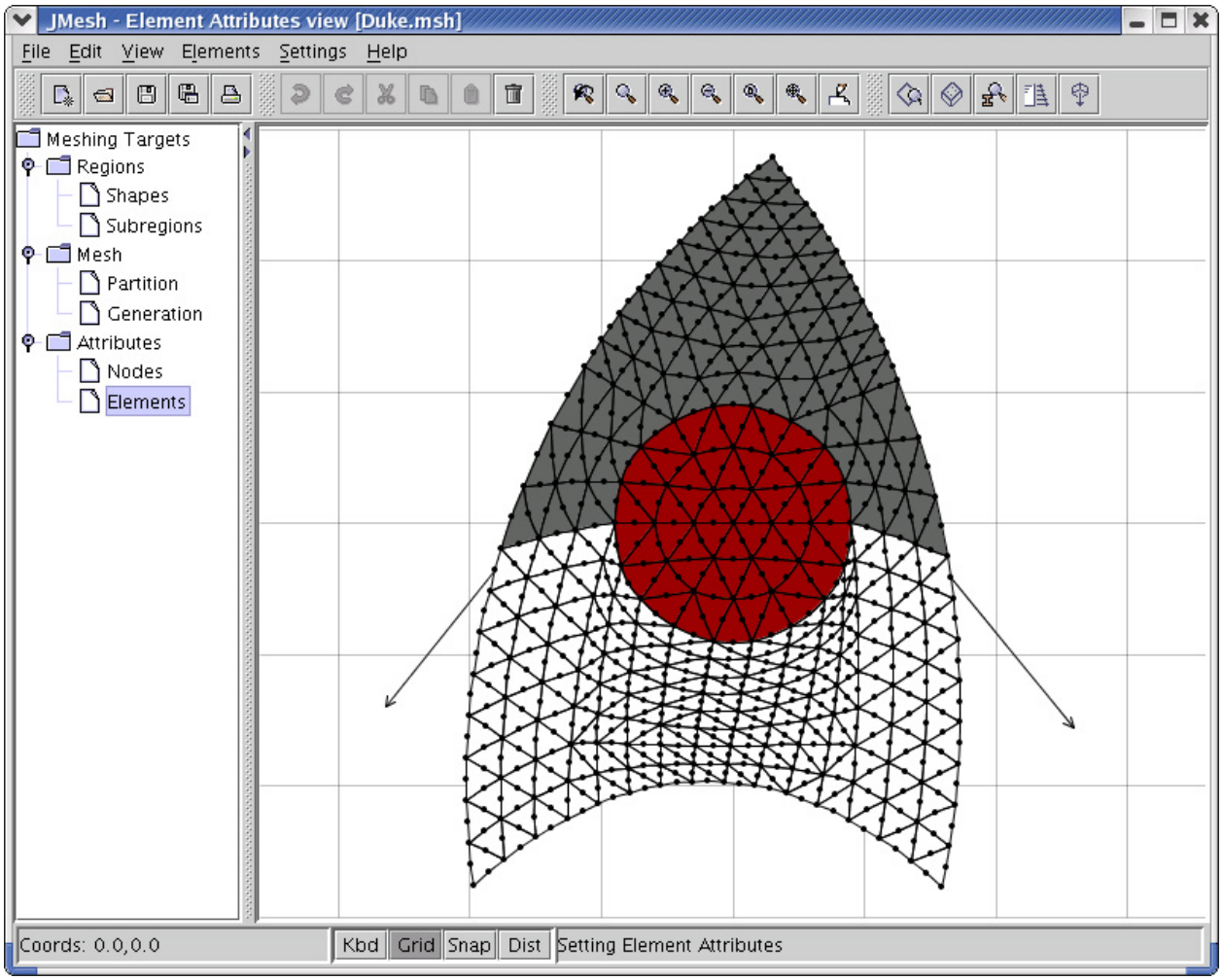


Figura 5.19: Duke, mascote Java

Capítulo 6

CONSIDERAÇÕES FINAIS

Ao longo do tempo, algumas iniciativas de desenvolvimento de *software* pela comunidade acadêmica resultaram em produtos dependentes de sistema operacional, pouco amigáveis, escritos em linguagens de programação não apropriadas, de expansão, distribuição e manutenção difíceis, desenvolvidos por equipes fechadas, com documentação deficiente, entre outras limitações. Tais fracassos podem ser creditados à falta de disposição da comunidade em se apropriar das tecnologias emergentes ou mesmo à inexistência das mesmas.

Esta constatação confronta-se com o surgimento e aprimoramento de soluções tecnológicas para desenvolvimento de *software*, como programação orientada a objetos, linguagem Java, XML (eXtensible Markup Language), padrões de projeto de *software*, entre outras. O domínio destes recursos e a aplicação dos mesmos no aprimoramento progressivo dos modelos, sem ter que recomeçar o processo a cada novo aperfeiçoamento, requer um ambiente computacional segmentado, amigável a mudanças e escalável em complexidade.

O `INSANE` é um projeto de desenvolvimento colaborativo de *software* que pretende utilizar as tecnologias emergentes para criar um sistema computacional para análise estrutural através de modelos discretos. Este projeto, criado junto com a dissertação aqui apresentada, já conta com o envolvimento de um (1) aluno de iniciação científica, cinco (5) alunos de mestrado e um (1) aluno de doutorado.

As principais contribuições desta dissertação, responsável por permitir o rápido envolvimento de todos esses alunos, bem como segestões para futuros trabalhos do projeto `INSANE`, são apresentadas a seguir.

6.1 Contribuições deste Trabalho

O gerador de malhas desenvolvido neste trabalho é o primeiro produto do sistema `INSANE`. Como conseqüência disto, o início de sua implementação foi marcado pelo desbravamento do grande número de tecnologias disponíveis, familiarização com as mais indicadas, utilização experimental destas no desenvolvimento do aplicativo, seleção e adoção das mais apropriadas.

O desenvolvimento dos novos segmentos do sistema `INSANE`, já em andamento, certamente incluirá as tecnologias emergentes, mas este desenvolvimento tem o suporte de uma plataforma de desenvolvimento consolidada. Assim, essa exploração inicial, com o objetivo de dominar as tecnologias disponíveis para desenvolvimento colaborativo de aplicações, talvez seja mais importante que o gerador de malhas obtido.

Entre as diversas tecnologias exploradas, pode-se destacar o universo da Programação Orientada a Objetos, a linguagem de programação Java (escolhida para implementar o sistema `INSANE`), a plataforma de desenvolvimento, composta pelo *J2SDK* (pacote de desenvolvimentos Java da Sun Microsystems, disponivel em `www.sun.com`), *Ant* (automatizador de processos de compilação, execução e distribuição da Apache *software* Foundation, sub-projeto Apache Jakarta Project, disponivel em `ant.apache.org`) e *Jedit* (editor de texto, de código aberto disponível em `www.jedit.org`) e os padrões de projeto de *software* (*MVC*, *Observer* e *Command*). Outras tecnologias que poderiam ser utilizadas ainda não foram incorporadas ao projeto `INSANE`. Entretanto, as possibilidades que os recursos tecnológicos para desenvolvimento de *software* oferecem, constituem amplo campo de pesquisa na área de métodos numéricos e computacionais aplicados à Engenharia.

A concepção da arquitetura lógica do *software* é outra importante contribuição deste trabalho. A adoção de uma arquitetura em camadas, formada pela combinação de vários padrões de projeto de *software* orientado a objetos, deu ao sistema características potenciais de expansão, modularidade, facilidade de manutenção e escalonamento de complexidade.

A exploração e domínio dos recursos gráficos interativos da plataforma Java (pacotes `javax.swing`, `java.awt`, classes `java.awt.Graphics` e `java.awt.Graphics2D`) permitiu o desenvolvimento de um aplicativo de fácil utilização e cuja implementação pode ser expandida

para disponibilização de novas facilidades.

Outro recurso relevante da plataforma Java, explorado e utilizado no desenvolvimento deste trabalho, foi o pacote `Collection`. O domíniio deste recurso permitiu a fácil manipulação das estruturas de dados da aplicação.

A utilização de recursos Java para criação e manipulação de arquivos *XML* (eXtensible Markup Language), linguagem de marcação que vem sendo adotada como padrão para troca de informações através da *WEB*, é outro aspecto relevante deste trabalho. Isto permitirá que, no futuro, com a mudançá da arquitetura física do sistema, o mesmo possa ser utilizado através da Internet.

Os algoritmos de geração de malhas implementados neste trabalho ilustram o potencial das novas tecnologias. Neste trabalho, as tradicionais rotinas de geração de malhas de elementos finitos triangulares de três e seis nós e quadrilaterais de quatro e oito nós foram substituídas por classes especializadas e flexíveis. Estas classes disponibilizam elementos Serendípticos quadrilaterais (com $4n$ nós, $n \in \aleph$) e triangulares (com $3n$ nós, $n \in \aleph$) e elementos Lagrangeanos quadrilaterais (com n^2 nós, $n \in \aleph$, $n > 1$) e triangulares (com $(n + n^2)/2$ nós, $n \in \aleph$, $n > 1$). Elementos triangulares podem ainda ser obtidos através da subdivisão de elementos quadrilaterais em suas diagonais.

Obteve-se um produto ainda em fase de aprimoramento, mas bastante superior aos obtidos com a utilização das tecnologias tradicionais no que diz respeito ao reuso, potencial de expansão, modularidade, facilidade de manutenção, uniformidade na estrutura da aplicação, incremento da padronização no desenvolvimento, aplicação imediata por outros desenvolvedores e redução da complexidade do sistema.

6.2 Sugestões para Trabalhos Futuros

A facilidade de expansão da aplicação desenvolvida neste trabalho possibilita a inclusão de novos recursos e aprimoramento dos existentes. Algumas sugestões para trabalhos futuros são apresentadas a seguir.

1. Estudo de estruturas de dados mais eficientes, compatíveis com modelos de elementos finitos e adequadas para os componentes de apresentação. Na versão atual do gerador de malhas, a estrutura de dados baseia-se em listas de elementos geométricos (instâncias de classes da *API Collection*). A implementação da estrutura de dados *Half-Edge* (Mäntylä 1988), já em andamento, corrigirá esta limitação do sistema.

2. Implementação do Mecanismo de Propagação de Mudanças do padrão *Observer* para dar suporte a um número variável de pares *Vista-Controlador*, possibilitando visualizar diferentes apresentações para um mesmo modelo e ativação simultânea de diferentes modelos.

3. Implementação da combinação de elementos finitos de diversos tipos (barras, placas e cascas, etc).

4. Implementação de novas classes para contemplar outras técnicas de geração de malhas como Avanço de Fronteira, Decomposição de Domínio e Delauney.

5. Implementação de novas classes que suportem modelos tridimensionais e expansão do código para permitir a visualização de tais modelos através da manipulação das classes do pacote *Java3D*.

6. Incorporação de novas tecnologias ao projeto `INSANE` como o *CVS* que facilita o desenvolvimento colaborativo e JUnit que automatiza os testes do sistema.

7. Inclusão de novas camadas físicas que suportem o uso de servidores de banco de dados e aplicações como *MySQL* (`www.mysql.com`), *Tomcat* (`jakarta.apache.org/tomcat`), *JBoss* (`www.jboss.org`), *JDBC* (`java.sun.com/products/jdbc`) e outros.

8. Implementar suporte à plataforma *J2EE* para utilização online da aplicação.

Referências Bibliográficas

Alvim, P. (2003), 'Open source: Os novos desafios de negócios e a indústria de ti', *Developers Magazine* (80), 13–15.

Brugiolo, M. A. & Pitangueira, R. L. (2004*a*), *INSANE - Manual de Desenvolvimento*, URL: `www.dees.ufmg.br/insane`, Departamento de Engenharia de Estruturas, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brasil.

Brugiolo, M. A. & Pitangueira, R. L. (2004*b*), *INSANE - Manual de Utilização*, URL: `www.dees.ufmg.br/insane`, Departamento de Engenharia de Estruturas, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brasil.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1995), *Pattern-Oriented Software Architecture: A System of Patterns*, Vol. 1, Wiley.

Camarão, C. & Figueiredo, L. (2003), *Programação de Computadores em Java*, LTC.

Deitel, H. M. & Deitel, P. J. (2003), *Java Como Programar*, 4ª edn, Bookman.

Flanagan, D. (2000), *Java: O Guia Essencial*, Editora Campus.

Fonseca, G. L. (1989), Geração de malhas através de mapeamentos transfinitos, Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

Fowler, M. & Scott, K. (2000), *UML Essencial: um breve guia para a linguagem-padrão de modelagem de objetos*, 2ª edn, Bookman.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns - Element of Reusable of Object Oriented Software*, Editora Addison-Wesley.

Goodrich, M. T. & Tamassia, R. (2002), *Estruturas de Dados e Algoritmos em Java*, Bookman.

Grand, M. (1998), *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, Vol. 1, John Wiley & Sons.

Gupta, S. (2004), 'The misteries of business object', URL: `javaboutique.internet.com/tutoriais/businessObject`. Última Consulta: 08/2004.

Horstmann, C. S. & Cornell, G. (2001*a*), *Core Java 2 - Fundamentos*, Vol. 1, Makron Books.

Horstmann, C. S. & Cornell, G. (2001*b*), *Core Java 2 - Recursos Avançados*, Vol. 2, Makron Books.

Jacobson, I., Christerson, M., Jonsson, P. & Övergaard, G. (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley.

Kreyszig, E. (1993), *Advanced Engineering Mathematics*, 7ª edn, John Wiley & Sons.

Larman, C. (2000), *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos*, Bookman.

Liesenfeld, R. (2002), 'Processando xml em java', *Java Magazine* (Edição 02), 48–52.

Lozano, F. (2003), 'Entenda a tecnologia xml', *Revista do Linux* (Edição 48), 42–49.

Lozano, F. (2004), 'Padrões e arquiteturas', *Java Magazine* (Edição 15), 18–19.

Mäntylä, M. (1988), *An Introduction to Solid Modeling*, Computer Science Press.

Miranda, A. C. O. (1999), Integração de algoritmos de geração de malhas de elementos finitos, Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

Pagliosa, P. A. (2004), 'Projeto de sistemas orientado a objetos', URL: `www.dct.ufms.br/~pagliosa/SystemAnalysis/ps1.pdf`. Departamento de Computação e Estatística, Universidade de Mato Grosso do Sul, Última Consulta: 09/2004.

Pietro, G. A. (2001), Utilização de padrões de projeto de software na reengenharia de sistemas, Master's thesis, Universidade Federal de São Carlos, São Carlos, SP, Brasil.

Rowe, G. W. (2001), *Computer Graphics with Java*, Palgrave.

Soriano, H. L. & Lima, S. S. (1999), *Método de Elementos Finitos em Análise de Estrutura*, Universidade Federal do Rio de Janeiro.