Plataforma para o Desenvolvimento de Ferramentas Baseadas em Diagramas

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Renato Tutumi e aprovada pela Comissão Julgadora.

Campinas, 23 de fevereiro de 1996.

Prof. Dr. Hans Kurt Edmund Liesenberg
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Tutumi, Renato

T889p Plataforma para o desenvolvimento de ferramentas baseadas em diagramas /Renato Tutumi -- Campinas, [S.P. :s.n.], 1996.

Orientador : Hans Kurt Edmund Liesenberg

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

1. Projeto de sistemas. 2. Engenharia de software auxiliada por computador. 3. Gráficos por computador. 4. Programação orientada a objetos. I. Liesenberg, Hans Kurt Edmund. II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Tese de Mestrado defendida e aprovada em 14 de fevereiro de 1996 pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a). Paulo César Masiero

Prof (a). Dr (a). Edmundo Roberto Mauro Madeira

Prof (a). Dr (a). Hans Kurt Edmund Liesenberg

Plataforma para o Desenvolvimento de Ferramentas Baseadas em Diagramas[1]

Renato Tutumi[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Hans Kurt Edmund Liesenberg[3] (Orientador)
- Paulo César Masiero[4]
- Edmundo Roberto Mauro Madeira[3]
- Luiz Eduardo Buzato[3] (Suplente)

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Ciência da Computação pelo Instituto de Ciências Matemáticas de São Carlos - Universidade de São Paulo.

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]Professor do Instituto de Ciências Matemáticas de São Carlos - Universidade de São Paulo.

Dedico este trabalho aos meus pais,
por estarem sempre iluminando meu caminho.
Onde quer que vocês estejam agora,
saberão da imensa gratidão e amor que sinto por eles.

Agradecimentos

Ao CNPq, pelo apoio financeiro recebido.

Ao meu irmão e grande amigo Roberto, pelo apoio e incentivo dado à minha formação pessoal e profissional.

Ao Prof. Hans, pela paciência em acompanhar e corrigir meus trabalhos.

Ao meu grande amigo Fábio, sua ajuda foi fundamental. As "duras" críticas e a dedicação nas correções deste trabalho me ajudaram a ser mais exigente com que escrevo.

Aos colegas de república, Galileu, Cacão, Mestre Léo e Hebão, pelo ambiente tão alegre e descontraído. Ao Fileto e Juliano, por aturarem meu mau humor nos momentos finais desta tese e pela convivência no dia-a-dia.

Aos meus amigos do DCC, em especial ao Rober, Raimundo, Nabor, Núccio, Flávio Lenz, Mivs, Karen, Helena, Maria Cláudia, Rosiane e muitos outros que ajudaram a fazer deste ambiente não só um lugar de trabalho, mas também um ponto de encontro de grandes amigos.

Aos meus amigos do CTI, que proporcionaram grandes alegrias tanto no aspecto profissional quanto pessoal. Vocês contribuíram significativamente para finalização deste trabalho. Agradeço especialmente à grande amiga Ana Luisa pelo incentivo quase que "diário."

Ao mundo enfim.

A satisfação está no esforço
e não apenas na realização final.
Gandhi

Resumo

A plataforma é uma base conceitual na forma de um conjunto de classes abstratas que visa apoiar a construção de ferramentas que realizam algum tipo de manipulação de diagramas, tais como editores gráficos e simuladores. O propósito é colaborar para que os resultados do esforço colocado no desenvolvimento de tais ferramentas possam ser mais facilmente estendidos e reutilizados. O trabalho desenvolvido consistiu na análise do domínio dessa família de aplicações, a fim de identificar as características comuns e modelá-las segundo o paradigma de objetos. Para auxiliar o emprego da plataforma, um procedimento de utilização também é definido.

Abstract

A conceptual framework based on a set of abstract classes is being proposed, that intends to aid the construction process of diagram manipulating tools, such as diagram editors and simulators. The major goal was to contribute to the achievement of a higher degree of extensibility and reusability of the results, produced by efforts put into the development of this kind of tools. A domain analysis was carried out in order to identify common features of this category of tools and to model them in accordance to the object paradigm. Guidelines for the use of the proposed set of abstract classes are provided as well.

Conteúdo

Capítulo 1

Introdução

O emprego de modelos abstratos é fundamental no desenvolvimento de sistemas mais complexos para a captura de suas características. Notações gráficas ou diagramas são comumente empregados na construção de modelos. A grande vantagem desse tipo de notação é o seu poder de expressão e a facilidade de sua compreensão. Entretanto, uma utilização mais extensiva de diagramas deve ser apoiada por ferramentas específicas, tais como editores e simuladores.

O presente trabalho é voltado para a especificação estrutural de tais ferramentas baseado em conceitos do paradigma de objetos. O motivo da escolha do paradigma será dado mais adiante de forma mais detalhada. Em síntese, a justificativa deve-se à preocupação em abranger uma grande variedade de ferramentas e diagramas. O paradigma de objetos tem-se mostrado adequado a esse propósito, pois facilita estender aplicações e aumenta a possibilidade de reutilização.

O resultado gerado é denominado **plataforma**. Este termo é mais comumente usado para designar uma determinada configuração de hardware. Entretanto, o sentido desejado é outro, ou seja, a plataforma serve como uma base conceitual para a modelagem das ferramentas mencionadas anteriormente. A plataforma foi desenvolvida segundo as recomendações do método OMT [RBP+91] e incorpora características consideradas genéricas tanto para ferramentas quanto para diagramas.

A experiência obtida no emprego do método OMT é relatada no decorrer da dissertação e pode ser útil para aqueles que já usam este método ou se interessam em conhecê-lo. O presente capítulo define o problema contemplado, descreve a importância e motivação do presente trabalho, estabelece as circunstâncias e o escopo no qual os resultados podem ser empregados e, ao final, apresenta uma visão macroscópica dos capítulos seguintes da dissertação.

1.1 Definição do Problema

Profissionais envolvidos no desenvolvimento de sistemas de computação fazem uso de modelos para facilitar o entendimento de um problema ou esboçar uma solução. Apenas um modelo, contudo, não é suficiente. É preciso também encontrar formas adequadas para melhor representá-lo. Mais precisamente em computação, a representação através de diagramas é uma prática comumente empregada.

Durante muito tempo, a construção de diagramas foi realizada empregando-se apenas "lápis e papel." Com o surgimento de ferramentas CASE, esta atividade passou a ser muito mais fácil e rápida, exigindo menos esforço de criação e alteração, além de automatizar muitas tarefas suscetíveis a erros, como a verificação de consistência envolvendo diversas entidades que compõem uma especificação não trivial.

Mesmo uma representação particular pode possuir variações que se estendem desde diferenças na forma de apresentação de seus componentes até na sua semântica. Em geral, as ferramentas são desenvolvidas especificamente para um conjunto predefinido de funções e diagramas. Se o usuário, por questões de adequação às suas particularidades, quiser uma variante da ferramenta ou do tipo de diagrama, as mudanças necessárias são penosas. Dar suporte para tais mudanças não é algo muito simples de se conseguir, principalmente em sistemas complexos, como é o caso do ambiente que será desenvolvido no Projeto Xchart [Lie93], do qual o presente trabalho faz parte. Diante desse quadro, o que se pode tentar é reduzir ao máximo os esforços requeridos para a extensão de ferramentas que manipulam diagramas. Também existe uma demanda de reutilização nos mais diferentes níveis, desde a especificação até o código gerado em alguma linguagem de programação.

1.2 Escopo do Trabalho

O assunto contemplado no presente trabalho é decorrente das dificuldades de construção e alteração das ferramentas usadas em computação, especificamente daquelas baseadas em diagramas.

O trabalho aqui descrito aborda esse problema e se concentra na parte estrutural destas ferramentas. A intenção foi encontrar um denominador comum entre os vários tipos de ferramentas e diagramas conhecidos.

Com base nesta idéia, nasceu a proposta de uma **Plataforma de Auxílio ao Desenvolvimento de Ferramentas Baseadas em Diagramas**. As Ferramentas Baseadas em Diagramas (FBDs) caracterizam-se por realizarem alguma espécie de manipulação sobre diagramas, tal como editar, simular a execução do comportamento,

fazer análises estruturais, mapear a representação para código em alguma linguagem de programação e assim por diante.

O escopo de FBDs está relacionado aos interesses do Projeto Xchart, o qual pretende construir um ambiente de ferramentas de auxílio a utilização da notação Xchart, derivada de estadogramas[1] [Har87a].

1.3 Emprego da Plataforma

O uso da plataforma não é destinado a usuários finais de ferramentas, mas aos responsáveis pelo seu desenvolvimento. Exemplos de situações em que o seu emprego é recomendado são:

- desenvolvimento de ambientes integrados de ferramentas, onde são empregadas várias ferramentas para manipular diversos tipos de diagramas. A vantagem é poder reutilizar as partes que são comuns a todas elas e, assim, dedicar mais esforços nas particularidades de cada caso.

- construção de ferramentas cuja representação gráfica utilizada esteja sujeita a alterações, tais como, adequação a um contexto específico e evoluções na notação. Neste caso, a extensibilidade é uma propriedade muito útil.

1.4 Objetivos

O objetivo do presente trabalho é modelar a plataforma de tal modo que a organização e o conteúdo de suas classes favoreçam a construção de ferramentas (do tipo FBD) mais fáceis de serem alteradas ou estendidas, além de serem capazes de fornecer componentes reutilizáveis para construção de outras ferramentas (também do tipo FBD).

É importante ressaltar que a plataforma não tem a pretensão de ser tão genérica a ponto de atender a uma infinidade de diagramas e ferramentas atualmente existentes, já que a diversidade de diagramas é muito grande e as suas características não obedecem um padrão bem definido.

O trabalho também relata a experiência na utilização do método OMT, descrevendo passo a passo as atividades de modelagem, baseado em um mesmo exemplo

[1] *statecharts*, neologismo já usado em outros trabalhos

durante todo o processo. O livro de Rumbaugh et al. [RBP+91] é bastante abrangente, mas carece de um único exemplo onde o processo é empregado.

1.5 Motivação

No princípio do trabalho, a idéia era realizar uma modelagem específica para ferramentas baseadas em estadogramas. Essa notação gráfica descreve o comportamento de sistemas reativos [HP85], estendendo a representação dos diagramas de transição de estados com: hierarquia, ortogonalidade e comunicação. Mais informações sobre esta notação são apresentados no Apêndice B. Estadogramas, em particular, são usados no exemplo de utilização da plataforma descrito no Capítulo 5.

As motivações para realização do trabalho foram:

- No Departamento de Ciência da Computação (DCC/Unicamp) encontra-se em andamento o Projeto Xchart. Os trabalhos já desenvolvidos envolvem pesquisas na área de engenharia de software, com ênfase em interfaces homem-computador, mais especificamente na síntese de sistemas reativos distribuídos em ambientes heterogêneos. No escopo do projeto está sendo definido uma notação baseada em estadogramas, cujo nome é o mesmo designado ao projeto. A notação[2] possuirá recursos próprios para especificar sistemas reativos potencialmente distribuídos. Visando construir um *ambiente flexível de ferramentas para Xchart*, seria preciso utilizar uma estrutura básica mais estável, que todos os produtos gerados tivessem em comum. Essa estrutura básica será provida pela própria plataforma, objeto do presente trabalho;

- Durante os últimos anos, grupos de pesquisa do DCC/Unicamp e do Instituto de Ciências Matemáticas de São Carlos (ICMSC/USP) têm-se dedicado à construção de ferramentas de auxílio ao uso de estadogramas para apoiar o desenvolvimento de sistemas reativos, tais como um editor gráfico [Bat91] e um simulador [Can93, For91] para estadogramas. Os esforços do DCC concentraram-se mais na construção de um ambiente para o desenvolvimento de sistemas reativos, onde o nível de abstração provido é o de estadogramas. Alguns resultados desses esforços são um gerador automático de código a partir de especificações em estadogramas [Fil91], posteriormente estendido [LL94], e estudos sobre o uso de estadogramas para descrever o controle de diálogo de interfaces [Luc93]. O grupo do ICMSC direcionou seus esforços para a validação de especificações

[2]Mais detalhes sobre o projeto e a notação podem ser encontrados em "http://www.dcc.unicamp.br/projects/Xchart/".

baseadas em estadogramas. Os esforços dos dois grupos contemplam abordagens complementares. Atualmente, estes ambientes atingiram um alto nível de sofisticação e complexidade, porém de difícil integração. Provavelmente pela falta de um planejamento inicial visando facilitar este tipo de tarefa. A plataforma poderá vir a facilitar a integração e a fatoração de esforços futuros, tanto na construção de novas ferramentas quanto na re-engenharia das já existentes.

É interessante também que o ambiente integrado de ferramentas do projeto possa manipular mais de um tipo de diagrama. STATEMATE [HLN+90a, iLI89], por exemplo, é um ambiente de desenvolvimento de sistemas reativos que permite utilizar três tipos de diagramas: Diagramas de Atividade (*Activity Charts*), Estadogramas e *Module Charts*.

1.6 Processo de Desenvolvimento

A modelagem da plataforma foi realizada conforme os princípios do paradigma de objetos. Este texto assume que o leitor esteja familiarizado com tal paradigma. Definições de termos como objeto, classe, herança e outros pertinentes ao paradigma podem ser encontrados em [RBP+91].

Inicialmente foi feito um estudo para verificar a adequação do paradigma de objetos, bem como o método de modelagem a ser empregado, aos objetivos estipulados.

Em seguida, foram coletadas informações, das quais seriam extraídas as características essenciais, ou gerais, dos diagramas comumente usados na representação de sistemas. Pode-se dizer que esta atividade configurou-se como uma *Análise de Domínio* [PD90, Hea90] de Diagramas, já que, de acordo com Prieto [PD90] "uma análise de domínio consiste no processo pelo qual as informações usadas no desenvolvimento de sistemas de software são identificadas, capturadas e organizadas, com o propósito de torná-las reutilizáveis na criação de novos sistemas."

Conforme o mesmo autor, contudo, "a análise de domínio é conduzida de uma maneira *ad-hoc* e histórias de sucesso são muito mais uma exceção do que regra. O processo de abstração dos conceitos, através da identificação das características em comum, é considerado uma atividade exclusivamente humana (i.e., inteligente) e associada a uma experiência" sendo o mecanismo de "ganhar experiência, um processo de aprendizado lento e não estruturado" já que "... o conhecimento de um domínio evolui naturalmente ao longo do tempo até que uma experiência razoável seja acumulada e vários sistemas tenham sido implementados, de modo que abstrações possam ser isoladas e **reutilizadas**." Portanto, "para formalizar o processo da análise de

domínio é preciso encontrar maneiras de se extrair, organizar, representar, manipular e entender a informação reutilizável."

O termo análise de domínio foi introduzido por Neighbors [Nei81] como "a atividade de identificar objetos e operações de uma classe de sistemas similares em um domínio particular de problema."

Muitos autores não se preocupam em "como fazer" a análise de domínio. McCain [McC85] é o primeiro a tentar atacar este problema, integrando o conceito de análise de domínio ao processo de desenvolvimento de software. McCain introduz também um conjunto de diretrizes para a condução da análise de domínio. Consiste em três passos básicos: identificação de entidades reutilizáveis, abstração (ou generalização), classificação e catalogação para posterior reutilização.

Por último, as características identificadas anteriormente foram mapeadas para modelos e representações, conforme definidos no método OMT. O resultado foi um conjunto de hierarquia de classes e relacionamentos. Cada hierarquia é composta por diferentes níveis de abstração. Nos níveis superiores têm-se as características essenciais. Por exemplo, informações sobre a topologia[3] de um diagrama (informação estrutural). Nos níveis inferiores, as informações mais específicas ou particulares como, características de apresentação dos componentes. Este último é um exemplo de informação que não é essencial, pois a semântica de um diagrama, via de regra, não depende fortemente de informações gráficas e geométricas, ao contrário das informações do primeiro exemplo.

1.7 Estrutura da Dissertação

Os capítulos da dissertação estão organizados conforme a seqüência cronológica das atividades realizadas. No Capítulo 2 faz-se uma descrição dos estudos realizados sobre diagramas. Essa atividade foi o ponto de partida para a identificação das características essenciais dos diagramas. No Capítulo 3 é tratada a questão dos métodos de modelagem orientados a objetos e as razões para a escolha do método OMT. O papel de um método no processo de desenvolvimento foi formalizar os passos a serem seguidos na modelagem da plataforma. O Capítulo 4 relata o processo de desenvolvimento da plataforma. O Capítulo 5 reforça a descrição dos resultados obtidos através de um exemplo de utilização da plataforma. Por último, no Capítulo 6 são apresentadas as conclusões do trabalho.

Segue-se uma descrição sucinta dos capítulos:

[3]Informações sobre a conectividade entre componentes do diagrama abstraída de detalhes espaciais.

Capítulo 1: Definição do problema abordado, proposta, objetivos e motivação do presente trabalho.

Capítulo 2: Revisão bibliográfica acerca dos estudos sobre diagramas e trabalhos correlatos ao executado.

Capítulo 3: Discussão sobre métodos de modelagem orientados a objetos e razões para escolha do método OMT.

Capítulo 4: Descrição do processo de desenvolvimento da plataforma.

Capítulo 5: Exemplo de utilização da plataforma.

Capítulo 6: Conclusões do presente trabalho, problemas encontrados, contribuições e sugestões para futuras pesquisas.

Apêndice A: Descrição da notação utilizada no método OMT: visa ajudar no entendimento dos modelos apresentados no Capítulo 4.

Apêndice B: Descrição de Estadogramas: visa ajudar no entendimento do exemplo de uso da plataforma descrito no Capítulo 5.

1.8 Comentários

No final de cada capítulo, a seção "comentários" cita as principais referências bibliográficas utilizadas na discussão do assunto abordado.

Capítulo 2

Revisão Bibliográfica

No capítulo anterior foi apresentada uma visão panorâmica do presente trabalho. Este capítulo descreverá os estudos que precederam o desenvolvimento da plataforma. O assunto aqui abordado é relacionado com **diagramas**. Os resultados de outro estudo preliminar são apresentados no próximo capítulo, trata-se do paradigma de objetos e dos métodos de modelagem orientados a objetos.

A Seção 2.1 relata sobre os estudos relacionados com a temática de diagramas. Inicialmente é enfatizada a relevância de tal recurso na modelagem de sistemas e são enumeradas algumas das propriedades desejáveis, no sentido de facilitar a sua utilização. Em seguida é apresentada uma bibliografia de trabalhos sobre representações gráficas. Os trabalhos contêm coletâneas de diagramas para diferentes tipos de contexto e sugestões para uniformização das notações. O propósito deste estudo foi identificar as características comuns aos vários tipos de diagramas e extrair as essenciais. Posteriormente, estas informações foram usadas na especificação da plataforma.

Na Seção 2.2 são comentadas publicações sobre o tema do trabalho desenvolvido.

2.1 Os Diagramas

"A diagram is worth a thousand words.
Many aspects of complex systems design
are much better stated in diagrams than in words."

Martin e McClure [MM85]

Os diagramas em questão são aqueles de uso exclusivo na representação de sis-

temas de computação. A frase de Martin e McClure expressa bem a importância dos diagramas diante de representações na forma de texto, assunto da Seção 2.1.1. No entanto, nem todos os diagramas são expressivos e de fácil uso. Neste sentido, a Seção 2.1.2 apresenta propriedades que, embora óbvias, não estão todas presentes nos diagramas considerados. A Seção 2.1.3 apresenta uma tentativa de classificação dos diagramas sob vários pontos de vista. Na Seção 2.1.4 é feita uma análise das características dos diagramas considerados na fase inicial de estudos e os padrões comuns a todos ou à grande maioria encontram-se descritos na Seção 2.1.5. Estes estudos iniciais foram essenciais à **Análise de Domínio** dos Diagramas.

2.1.1 Representações na Forma de Texto e Gráficas

Acredita-se que a capacidade de pensar dos seres humanos é influenciada pela linguagem utilizada para expressar as idéias [MM85]. Um exemplo desta crença é o seguinte: antigamente, quando a numeração usada era apenas a romana, a maioria das pessoas não era capaz de realizar operações aritméticas mais complexas como a divisão e a multiplicação. Somente após a difusão da numeração arábica é que tais operações tornaram-se amplamente usadas. Traçando um paralelo entre este fato e o processo de desenvolvimento de sistemas, a representação utilizada para especificar um problema e projetar sua solução pode ser um fator importante no esforço de desenvolvimento e qualidade do produto final. Uma representação que proíba a utilização de construções não estruturadas já é uma forma de influenciar positivamente na qualidade do software.

Esta seção procura enfatizar a importância das representações gráficas em um ambiente computacional e destaca suas vantagens em relação às representações na forma de texto. As razões são muitas. Uma delas é o fato da mente humana ser fortemente orientada a elementos visuais. Por isso, a taxa de informação transmitida por meio de figuras é geralmente maior do que por um texto [Tri88]. A utilização de elementos visuais é tão importante que até mesmo durante a leitura de um livro as pessoas costumam criar mentalmente as imagens da série de acontecimentos. Estes argumentos não têm o objetivo de desmerecer as representações na forma de texto, afinal ambas possuem seus méritos e não convém advogar o uso de um em detrimento de outro.

Considerando fatores humanos, algumas observações acerca do poder de expressão das representações gráficas diante de textos são apresentadas a seguir [Rae85]:

Acesso aleatório. A visão humana possui mecanismos tão sofisticados, que é capaz de ter acessos instantâneos e aleatórios a qualquer parte de um gráfico, enfocando regiões globais ou locais. Caso existam padrões previamente conhecidos, a

identificação das informações pode ser ainda mais rápida. Por outro lado, a leitura de um texto é essencialmente seqüencial, com exceção das buscas por palavras ou conjunto de palavras ao longo de um texto, cujo acesso passa a ser aleatório. Neste caso, a agilidade da busca deve-se à utilização de mecanismos próprios das representações gráficas dos símbolos que compõe um texto.

Múltiplas dimensões. Os textos são fluxos uni-dimensionais de dados. Os gráficos, por sua vez, podem ter duas ou mais dimensões expressas através de propriedades como forma, cor, textura, posição, direção e dimensão geométrica.

Taxa de transferência de dados. As informações contidas em gráficos podem geralmente ser mais rapidamente reconhecidas e absorvidas do que por meio de textos. Exemplos comprovados deste fato são as placas de sinalização, cartazes de propaganda, mapas rodoviários e outros.

Independência de linguagem. Mesmo havendo culturas e linguagens completamente diferentes, algumas representações gráficas podem ser entendidas sem necessidade de tradução, simplesmente associando elementos da realidade com os contidos na representação.

Atribuição de nomes. Apontar para objetos visíveis é uma prática comum quando se quer referenciá-los. Uma criança recorre a este recurso quando quer se expressar, mas ainda não é capaz de fazer isto através de palavras. O mesmo pode ocorrer com relação a elementos de um gráfico. Outra maneira de se fazer referências é através de um nome (referência indireta), neste caso não é preciso apontar para o objeto, esteja ele presente ou não, porém obrigam a introdução de um nível adicional de informação.

Existem casos em que a representação textual é mais adequada como, por exemplo, em fórmulas matemáticas e teoremas. A combinação das duas formas de representação tem sido uma prática comumente adotada com o objetivo de se usufruir das vantagens de ambas.

2.1.2 Propriedades Desejáveis aos Diagramas

Os diagramas possuem certas propriedades que evidenciam seu uso potencial. Para ser um "bom" diagrama, é importante que ele [Dav88]:

- ajude no processo de raciocinar sobre o problema; eventualmente pode ser um empecilho;
- possa ser manipulado (construído e impresso) com auxílio do computador;

- seja de fácil compreensão, podendo inclusive ser um meio de comunicação com o cliente, de modo que ele possa entender e interferir mais ativamente na construção de um modelo;
- possua construções com significado óbvio, ou seja, uma analogia com elementos conhecidos é suficiente para entendê-lo;
- possa ser subdivido em módulos e deste modo possibilitar a distribuição e estruturação dos elementos do diagrama mais adequadamente;
- permita sua visualização sob diferentes níveis de abstração (geral ou detalhada);
- ambigüidades, inconsistências e completitude possam ser verificadas automaticamente;
- possa ser mapeado (compilado) para código executável;
- possa ser usado em aplicações simples e complexas.

2.1.3 Diagramas Analisados

Inicialmente, este trabalho pretendia abordar somente uma família específica de diagramas, chamados **estadogramas** (veja Seção 1.5). Neste sentido, foram analisados os seus aspectos léxico, sintático e semântico, visando modelá-los em termos de objetos. No decorrer dos estudos, decidiu-se que uma abordagem mais genérica seria mais interessante, pois em um ambiente integrado de ferramentas é comum manipular mais de um tipo de diagrama.

Devido à grande diversidade de diagramas e à falta de um padrão bem definido entre eles, o primeiro passo foi no sentido de se identificar suas características e, assim, selecionar as mais genéricas. Em seguida, foi efetuada uma classificação dos vários diagramas considerados. A classificação fornece uma visão macroscópica dos diferentes tipos de diagramas e das características comuns entre eles.

A escolha dos diagramas submetidos a uma análise baseou-se em trabalhos já publicados, que contêm coletâneas de diagramas com enfoque em algum tema, comparações, alternativas de classificação e até sugestões para uma padronização. Com isso, obteve-se uma diversidade maior de amostras sem a necessidade de se analisar uma quantidade excessiva. A seguir é apresentada a descrição dos trabalhos encontrados.

Coletânea de Notações Gráficas para Projeto de Programas. O enfoque dado por Tripp em [Tri88] é voltado para diagramas usados no Projeto Estruturado

de Sistemas, mais especificamente ao fluxo de controle de programas. A coletânea inclui notações que surgiram desde 1977. Não existe diferenças significativas, com relação ao conjunto de construções, entre as notações. Na Figura 2.1 são ilustradas as construções básicas: seqüência, seleção (`if then else` e `case`) e repetição (`while` e `repeat`). As caixas T1,...,Tn representam as tarefas.

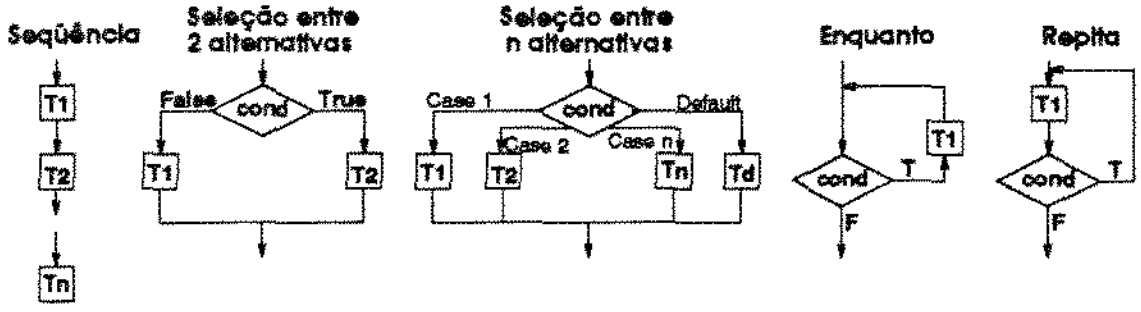


Figura 2.1: Construções básicas para fluxo de controle de programas.

Algumas notações possuem construções adicionais para representar paralelismo de processos, declarações de variáveis, abertura de arquivo e outras. Atualmente, a maioria delas é pouco conhecida, pois são bastante antigas e caíram em desuso, mas o fato que se quer ressaltar é a grande semelhança de uma notação para outra, principalmente no conjunto básico de construções. A diferença mais evidente é no aspecto de apresentação. Baseado nisto, Tripp classifica as notações gráficas coletadas por ele em 3 categorias:

- *box and line*: Rothon Diagram [Bro83], Ferstl Chart [Fer78], Problem Analysis Diagram (PAD) [FKHT81], Compact Chart [HS80], Software Diagram Description (SDD) [KS80], Diagrama de Estrutura [Chy84], Doran Chart [DT72], Schematic Logic [JT79], Greenprint [BEP80], Universal Flowcharter [HNRT79], Graphical Structured Flowchart [Rob81], FPL [TCU86];
- *box*: Flowblocks [Gro77], Boxchart [Jon87], Lindsey Chart [Lin77], SPDM Diagram [Mar79], FP Diagram [Pag87];
- *line*: Dimensional Flowchart [Wit77].

Devido à semelhança que existe entre as várias notações apresentadas, Tripp recomenda a adoção de uma delas como padrão, preferencialmente Flowchart (Figura 2.1). O mesmo se aplica para as notações gráficas com outros enfoques, ou seja, identificar as principais notações de cada área de uso e promover esforços no sentido de torná-las um padrão no nível internacional [Tri88].

Notações Gráficas para Programas. O artigo de Jonsson [Jon89] é uma crítica ao trabalho de Tripp [Tri88]. Jonsson destaca a omissão de notações gráficas precursoras às apresentadas por Tripp, tais como Jackson [Jac75], Warnier [War74] e Nassi-Schneiderman [NS73].

O autor propõe uma forma de classificação das notações tomando como base a *estrutura hierárquica* dos programas estruturados:

- *árvore*: Doran Chart [DT72], Warnier [War74], Jackson Structure Diagram [Jac75];
- *cascata*: Dimensional Flowchart [Wit77], Ferstl Chart [Fer78], Rothon Diagram [Bro83], Graphical Structured Flowchart [Rob81]; D-Chart [Bot86];
- *aninhamentos*: Nassi-Schneiderman [NS73], Lindsey Chart [Lin77], Boxchart [Jon87].

De acordo com Jonsson, a classificação de Tripp não retrata adequadamente a "lógica da representação." Na Figura 2.2 três formas diferentes e equivalentes de representação das estruturas hierárquicas são ilustradas.

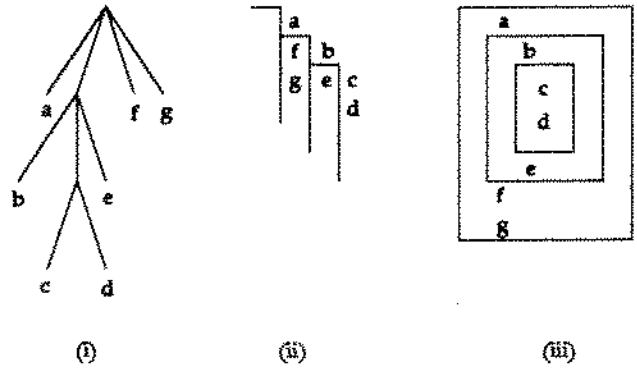


Figura 2.2: Modos de representação para estruturas hierárquicas.

Comparação de Técnicas para Especificação do Comportamento de Sistemas. A proposta de Davis [Dav88] limita-se ao aspecto comportamental dos sistemas. Descreve sucintamente e compara várias técnicas. A classificação das técnicas baseia-se nas formas de representação:

- *textual*: Linguagem Natural, Program Design Language (PDL) [CG75];
- *tabular*: Tabela de Decisão [Chv83], Matriz de Transição de Estados;

- *gráficas*: Diagrama de Transição de Estados [Bra77], Árvore de Decisão [Mor82], Structured Analysis/Real Time (SA/RT) [War86], Statecharts [Har87a], Requirements Engineering Validation System (REVS) [DV77], Requirements Language Processor (RLP) [Dav78], Specification and Description Language (SDL) [RS82], Process-oriented Applicative and Interpretable Specification Language (PAISLey) [Zav82], Petri-nets [Pet77].

Técnicas Diagramáticas para Análise, Projeto e Programação Estruturada. O livro de Martin e McClure [MM85] é o trabalho que mais contribuiu com os objetivos do presente trabalho. Os autores consideram muito importante o conhecimento das várias técnicas existentes e orientam sobre como escolher um bom conjunto de técnicas, dependendo das necessidades particulares. Os projetos, desde os mais simples até os de grande porte, precisam escolher adequadamente as técnicas que irão usar de modo que não sejam incompatíveis, redundantes ou conflitantes umas com as outras. Para isto, vários diagramas são descritos e classificados segundo uma de oito áreas de atuação. Na Figura 2.3, as áreas de atuação e as técnicas[1] consideradas são exemplificadas. Idealmente, as diversas técnicas deveriam convergir para conjuntos reduzidos, consistentes e coerentes de técnicas que se tornarão a base para projetos. Nesse sentido, o livro contém várias sugestões para a padronização dos elementos pertencentes a diagramas. Algumas dessas sugestões visam a facilitar a manipulação deles por computadores, pois a tendência observada pelos autores é a crescente automatização do processo de desenvolvimento de sistemas. Atualmente, essa tendência é uma realidade. Para tal basta observar a popularização de termos como Computer Aided System Analysis (CASA), Computer Aided Programming (CAP), Computer Aided Software Testing (CAST) e outros.

No lado esquerdo da figura estão as técnicas com enfoque nos *dados* e no direito com enfoque nas *atividades*. Como se vê, é possível que a mesma técnica esteja associada a mais de uma área. As quatro camadas representam os diferentes níveis de abstração. Dependendo da camada em questão, as atividades podem ser funções, processos ou procedimentos. O *nível 1* é o mais abstrato e compreende as técnicas que capturam a visão estratégica de como dados/funções são compostas. O *nível 2* refere-se à relação lógica entre dados/processos. Nos *níveis 3* e *4* a abordagem é sob o ponto de vista estrutural global ou detalhado dos programas, respectivamente.

Dois resultados foram gerados a partir da observação dos vários diagramas coletados: (1) um gráfico contendo três pontos de vistas para classificação das técnicas (Figura 2.4); e (2) um conjunto de características ou elementos comumente encontrados nos diagramas discriminados na próxima seção.

[1] HIPO = Hierarchical Input-Process-Output; HOS = Higher-Order Software.

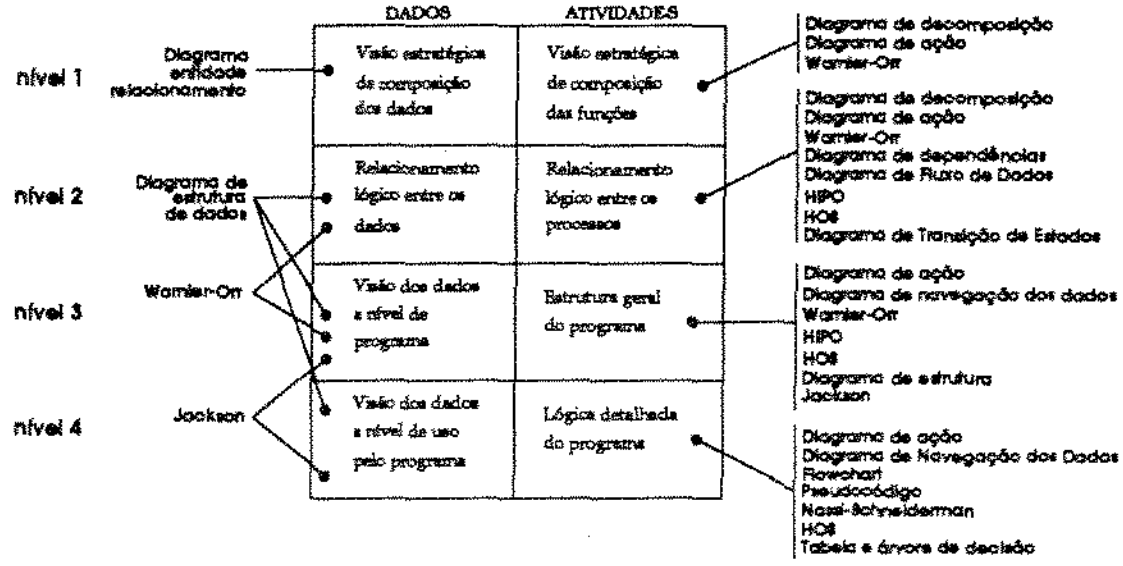


Figura 2.3: Áreas de atuação com suas respectivas técnicas.

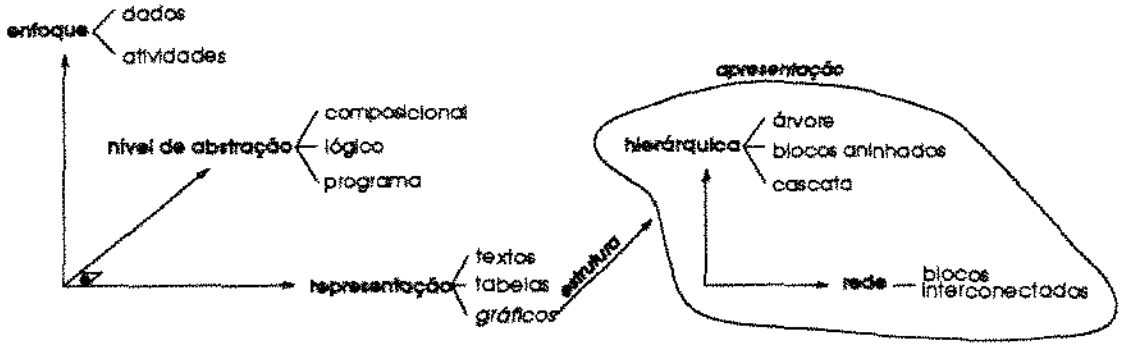


Figura 2.4: Classificação das técnicas coletadas.

Os três eixos da Figura 2.4 foram concebidos de modo a ilustrar formas independentes de se observar as técnicas. O eixo vertical **enfoque** considera o tipo de informação que é capturada pela técnica. As ramificações não são exclusivas. No eixo horizontal **representação** é considerada a forma de representação das informações, que pode ser por meio de: textos, tabelas ou gráficos. No presente trabalho, o interesse maior é nas representações gráficas, cuja estrutura organizacional e forma de apresentação dos seus componentes é ilustrada na região delimitada no lado direito da figura. Muitos dos diagramas para a descrição do fluxo de controle de programas estruturados apresentam uma estrutura hierárquica. Basta observar os exemplos de Tripp [Tri88] e Jonsson [Jon89]. Por outro lado, os relacionamentos entre dados enquadram-se naturalmente em estruturas em rede [MM85]. O eixo inclinado **nível**

de abstração tem o mesmo significado dos níveis de abstração propostos por Martin e McClure (Figura 2.3). O nível *composicional* refere-se aos elementos que compõem um dado ou atividade. O nível *lógico* refere-se à relação lógica entre dados ou atividades. O nível *programa*, por outro lado, refere-se às informações mais detalhadas sobre o uso de dados ou a estrutura de um programa.

2.1.4 Características dos Diagramas

Esta seção apresenta as características observadas nos diagramas citados anteriormente. As características foram reunidas em três grupos: componentes, estrutura e apresentação.

Componentes

O termo elemento designa qualquer objeto pertencente a um diagrama, como um rótulo, um símbolo gráfico ou um segmento de reta. Um componente é formado por um conjunto de elementos e um diagrama é formado por um conjunto de componentes. Comparativamente, os componentes são unidades de informação de mais alto nível que os elementos.

Exemplos de componentes podem ser extraídos do eixo **nível de abstração** da Figura 2.4. Os exemplos foram divididos em dois grupos:

1. dado, atividade, módulo, função, processo, procedimento, estado, comando (case, if, while, repeat), entidade; e
2. fluxo de dados, fluxo de controle, transição, associação, relação.

Estes dois grupos receberam uma denominação mais genérica, a ser apresentada na próxima seção.

De acordo com a observação dos vários diagramas, muitos dos componentes do grupo 1 e 2 possuem um nome ou texto associado, que servem para identificar e distinguir um componente do outro. Alguns componentes do grupo 2 possuem atributos como: orientação (fluxos e transições) e multiplicidade (associações 1:1 ou M:N).

Estrutura

Este tópico refere-se à estrutura organizacional dos componentes. Os diagramas analisados obedecem a uma estrutura na forma de grafo, com variações do tipo: ser cíclico ou acíclico e ser dirigido ou não.

Outra forma de se agrupar diagramas é segundo estruturas em rede ou hierárquicas. Analisando diagramas pelo ponto de vista do eixo **nível de abstração** da Figura 2.4, observou-se que no nível composicional a predominância é de diagramas com estruturas hierárquicas. Nos outros níveis, os diagramas com enfoque em dados apresentam uma estrutura em rede e os com enfoque em atividades apresentam uma estrutura hierárquica. Segundo Martin e McClure [MM85], os modelos de dados e os fluxos de dados são representados de forma mais natural através de estruturas em rede. Por outro lado, as estruturas hierárquicas são mais freqüentes em programas estruturados. Alguns defensores do método estruturado recusam-se a aceitar a existência de algo que não seja hierárquico e conseqüentemente, dão pouca importância ao modelo de dados. Nos métodos mais recentes, o enfoque a dados passa a ser bem mais evidente.

Apresentação

Como diagramas são uma representação gráfica, o aspecto visual também merece atenção. A análise desse aspecto consistiu em coletar os símbolos gráficos freqüentemente usados em diagramas. As figuras abaixo exemplificam como algumas construções de diagramas são representadas graficamente.

A Figura 2.5 ilustra componentes do grupo 1. Diagramas com um enfoque em dados e atividades representam esses componentes como retângulos e figuras geométricas arredondadas (elipses, retângulos com bordas arredondadas, círculos), respectivamente.

Figura 2.5: Representação gráfica para *dado* e *atividade*.

A Figura 2.6 ilustra exemplos de componentes do grupo 2. Normalmente são segmentos de reta ou curvas, interligando os componentes do grupo 1. Por exemplo, os fluxos definem a comunicação entre processos e as associações estabelecem relações entre entidades. Certos componentes do grupo 2 podem ser dirigidos ou não.

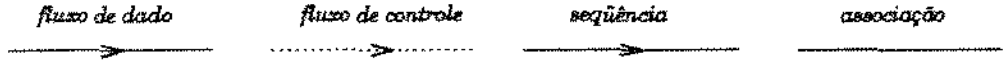


Figura 2.6: Representação gráfica para fluxos e relações.

Outro tipo de construção encontrada nos diagramas analisados é a multiplicidade na relação entre componentes do grupo 1. Isto é, a quantidade de instâncias de um componente que pode estar associada a quantidade de instâncias de outro tipo

de componente ou até do mesmo tipo de componente. Esta quantidade pode ser: zero, uma ou várias instâncias. Um exemplo de como isto pode ser representado graficamente está ilustrado na Figura 2.7. A notação utilizada é a de Diagramas Entidade-Relacionamento [MM85]. Quando não existe uma indicação de multiplicidade, assume-se a *default*, que normalmente é 1-para-1.

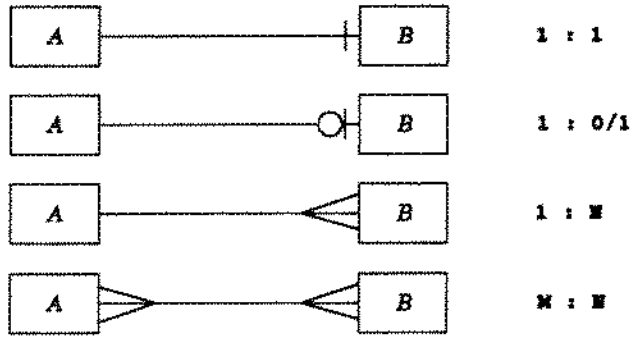


Figura 2.7: Representação gráfica para multiplicidade.

Relacionamentos entre componentes do grupo 1 também podem ser mutuamente exclusivos ou inclusivos, ou seja, um componente tem relação com apenas um (exclusão mútua) ou com todos os componentes de um grupo (inclusão). Na Figura 2.8 é ilustrado como esses dois casos são representados em Diagrama de Dependências [MM85]. No lado esquerdo da figura, o Processo A é sucedido pelo Processo B **ou** C. No lado direito, o Processo A é sucedido pelos Processos B **e** C.

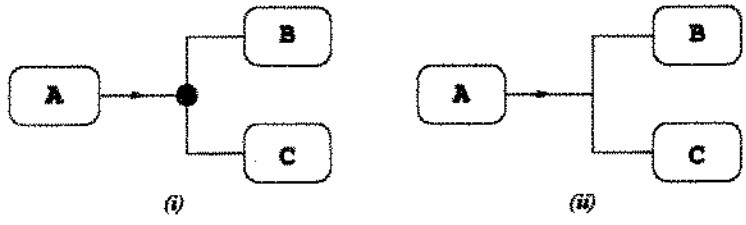


Figura 2.8: Representação gráfica para exclusão mútua (i) e inclusão (ii).

Alguns diagramas utilizam cores para destacar itens de especial importância, direcionar a atenção do observador, distinguir entre áreas ou blocos de tipos ou assuntos diferentes, simplificar diagramas complexos (apenas itens com a mesma cor são considerados a cada instante), sobrepor desenhos e assim por diante. Por exemplo, em um Diagrama de Decomposição [MM85], as atividades com diferentes níveis de detalhamento são destacadas com tonalidades de cinza, conforme ilustrado na Figura 2.9. A eficiência desse recurso depende dos critérios adotados. O uso com propósitos apenas estéticos não é recomendado.

Figura 2.9: Diferentes tonalidades de cinza podem simular o uso de cores.

O aninhamento de componentes é permitido em alguns diagramas. Não se trata de uma construção em especial, apenas uma forma diferente de se **apresentar** o mesmo diagrama. O objetivo é facilitar o entendimento, ocultando detalhes que não são relevantes em um determinado nível de abstração. Estes detalhes são revelados quando necessários. Essa técnica é muito útil, principalmente, em grandes diagramas. A Figura 2.10 mostra um exemplo simples de Diagrama de Fluxo de Dados antes e depois da "explosão" do Processo 2.

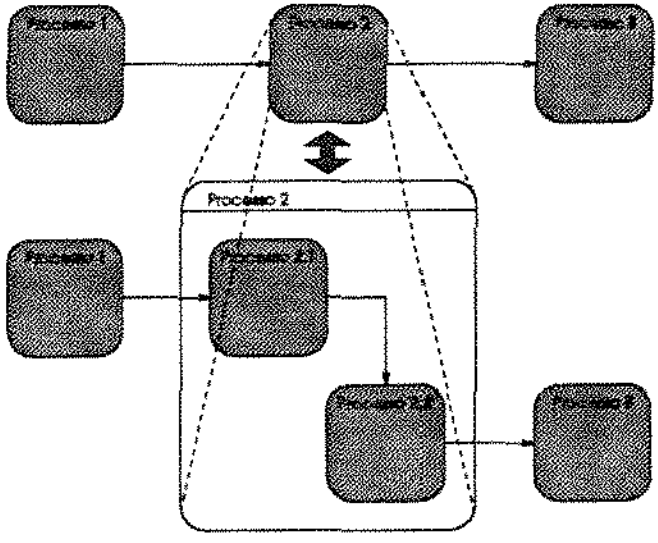


Figura 2.10: Representação gráfica do aninhamento de processos.

2.1.5 Características Essenciais

A partir das características levantadas na seção anterior, algumas características essenciais foram identificadas. Nesta tarefa, apenas os aspectos léxicos e sintáticos foram considerados. A semântica não é contemplada, pois é um aspecto muito dependente das particularidades de cada diagrama.

Dentre os Componentes citados anteriormente, os que pertencem ao grupo 1 passarão a ser chamados de **blocos** e os que pertencem ao grupo 2, de **conectores**. O

fato de um bloco ser um dado ou uma atividade é desconsiderado, pois trata-se de uma informação relevante apenas do ponto de vista semântico.

Os vários elementos levantados são considerados como simples **atributos** dos componentes bloco e/ou conector. Mas, quais seriam os atributos essenciais? Aqueles que estão presentes em todos os diagramas? A resposta é não. Se o critério fosse este, então nenhum atributo seria considerado essencial, pois sempre existem exemplos de diagramas que não possuem alguns dos elementos comumente utilizados. Então, qual o critério a ser adotado? O critério que pareceu ser mais apropriado foi o da freqüência de ocorrência de atributos em diagramas de tipos diferentes. Para os blocos, o atributo que aparece na maioria dos diagramas é o seu nome (nome do estado, da atividade, do processo) ou o próprio conteúdo (blocos de código fonte). Para os conectores são mais freqüentes os rótulos e a orientação (para distinguir a origem e o destino da conexão). Alguns atributos menos freqüentes dos conectores são os indicadores de multiplicidade (1:1, M:N) e de exclusão mútua/inclusão entre grupos de blocos.

Com relação à Estrutura de diagramas, ela está relacionada à sua sintaxe. As estruturas mais encontradas são a **hierárquica** e a em **rede**. Para estes dois casos as estruturas mais recomendadas para organização dos componentes são árvores e grafos, respectivamente.

Por último, a Apresentação, parte não essencial do diagrama, requer uma biblioteca de símbolos gráficos para desenho dos seus componentes. Os blocos são representados graficamente como figuras geométricas (retângulo, elipse, triângulo) com dimensão fixa ou variável, e com diferentes cores ou tonalidades de cinza. Os conectores, por sua vez, são representados graficamente como arcos, curvas ou segmentos de reta. Quando existe alguma regra de disposição dos blocos, as formas mais freqüentes são: da esquerda para direita (Árvore de Decisão [Mor82]) e de cima para baixo (Diagrama de Estrutura [YC78]). Diagramas com estrutura hierárquica podem ser apresentados como árvores (JSD [Jac75]), blocos aninhados (Nassi-Schneiderman [NS73]) ou em cascata (Ferstl Chart [Fer78]). No caso de estruturas em rede, a forma de apresentação é uma rede de blocos interligados (Diagrama de Transição de Estados [Bra77]).

2.2 Trabalhos Correlatos

Na procura por trabalhos correlatos foi difícil encontrar algum que pudesse contribuir significativamente na modelagem da plataforma ou com características semelhantes às do presente trabalho.

Por exemplo, o ICMSC/USP vem desenvolvendo ferramentas de apoio à utilização

de estadogramas. Elas são algumas das ferramentas às quais a plataforma deve auxiliar a construir e cujas características foram contempladas durante a análise das FBDs.

O Unidraw [VL90] é um caso particular de FBD. Trata-se de uma ferramenta que permite a edição de diferentes tipos de diagramas, bastando que se defina os objetos gráficos de interesse. Entretanto, o Unidraw limita-se apenas à edição. No caso da plataforma, o objetivo é ser mais abrangente.

Cabral et al. [CJCC93] relata em seu artigo a utilização do método OMT na modelagem de um editor de Diagramas de Objetos (representação usada para descrever o Modelo de Objetos do método OMT). Os modelos definidos por Cabral et al. influenciaram na modelagem da plataforma e também na escolha do método a ser usado.

O EDGE [PT90] considera os diagramas como uma coleção de nós e arestas. Este conceito é usado na definição estrutural do diagrama.

Mais informações sobre estes trabalhos são dadas a seguir.

2.2.1 ICMSC/USP

Ao longo dos últimos anos, o ICMSC tem acumulado experiências no estudo de estadogramas, através da construção de ferramentas úteis à validação de especificações baseadas nesta notação.

As ferramentas do ICMSC apresentam interface gráfica padrão Openlook. Existe um alto grau de integração entre elas, pois os dados produzidos por uma ferramenta podem ser utilizados por outra. Algumas das ferramentas em questão são:

- editor gráfico de estadogramas;
- simulador da execução de estadogramas. Opera em diferentes modos: por disparo de eventos selecionados manualmente de uma lista; por disparo de eventos contidos em um arquivo; e de forma programada, onde os eventos são gerados de acordo com distribuições de probabilidade especificadas pelo usuário;
- verificador de inconsistências nas especificações, tais como situações de *deadlock* e estados inatingíveis em uma simulação.

2.2.2 DCC/Unicamp

No Departamento de Ciência da Computação (DCC/Unicamp) também foram desenvolvidas ferramentas baseadas em estadogramas, porém os esforços se concentraram na construção de um ambiente de desenvolvimento de sistemas reativos, com ênfase na área de interfaces homem-computador [Luc93]. Uma das ferramentas transforma especificações na forma de estadogramas, complementadas com ações descritas em código C, em programas funcionalmente equivalentes [Fil91]. Uma proposta mais recente gera apenas dados para um interpretador, sem requerer o uso de um compilador, pois o código não é gerado [LL94].

Está planejado para o ano de 1996 a implementação do código que "executa" especificações em Xchart. Nesta abordagem, dados equivalentes a uma descrição em Xchart são produzidos por um compilador integrado a um editor gráfico de Xchart. Este código, ou sistema de execução de Xchart, é o responsável por "animar" a especificação, conforme eventos gerados no ambiente. Mais detalhes podem ser obtidos em [LLB95]. A descrição de Xchart e de sua semântica operacional formal podem ser obtidos, respectivamente, em [LL96] e [LHL96].

2.2.3 Vlissides & Linton [VL90]

Vlissides e Linton são os responsáveis pelo **Unidraw**, uma plataforma para a construção de editores gráficos em vários domínios, desde desenho artístico e composição musical até projetos de circuitos integrados e sistemas de computação. A arquitetura do Unidraw simplifica a construção dos editores de diagramas, fornecendo abstrações comuns entre os vários domínios. Estas abstrações são mapeadas em uma organização orientada a objetos composta por quatro hierarquias de classes:

- *componente*: representam os elementos do domínio. Encapsulam informações lógicas e de apresentação dos elementos. Nas suas subclasses são armazenadas as informações de relevância ao domínio.
- *representação externa*: define a "fórmula" de mapeamento do diagrama editado e a sua representação em arquivo.
- *ferramenta*: permite a manipulação direta dos componentes durante a edição de um diagrama. Possui recursos que visam dar ao usuário a percepção de estar manuseando diretamente os objetos em construção.
- *comando*: define as operações sobre os componentes e outros objetos. São semelhantes às mensagens que circulam entre objetos nos sistemas orientados

a objetos. Comandos podem mudar os valores internos de um componente, executar algum cálculo e assim por diante.

Conforme os autores, o Unidraw dá suporte ao desenvolvimento de editores gráficos para uma grande variedade de domínios, produzindo resultados com desempenho e recursos comparáveis aos editores comumente encontrados. Além de reduzir o tempo e esforços necessários à realização desta tarefa, pois a quantidade de decisões a serem tomadas é significativamente menor.

Um conceito que vale destacar é a distinção entre as informações de um componente em *subject* e *view*. O primeiro refere-se às informações que caracterizam o objeto em questão, e o segundo às informações de apresentação em um determinado contexto. A Figura 2.11 ilustra este conceito. O `Component Subject` da direita está associado a dois contextos diferentes: um sob o ponto de vista do `Editor1`, outro pelo `Editor2`. No nível `Subject` as informações relevantes são o tipo da porta lógica e suas conexões de entrada e saída. No nível `Viewer` estão as informações gráficas para o desenho da particular porta lógica. O nível `Viewer` corresponde à visualização do componente acrescido de operações sem influência sobre os componentes, tais como *scroll* e *zoom*. No nível `Editor` estão disponíveis as operações de manipulação dos componentes e gerenciamento das diferentes formas de visualização. No topo da figura está o Unidraw, que dá suporte à criação dos vários editores, os quais fornecem múltiplas visões de componentes.

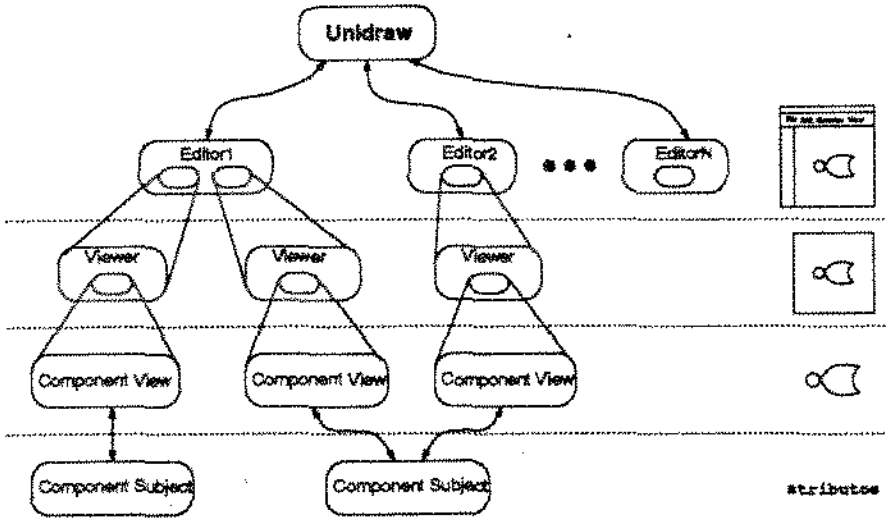


Figura 2.11: Estrutura geral dos editores baseados no Unidraw.

2.2.4 Cabral et al.[CJCC93]

Cabral **et al.** empregou o método de modelagem OMT no desenvolvimento de uma editor gráfico para representação do Modelo de Objetos do método OMT (veja Seção 3.2.1), ou seja, para a edição de Diagramas de Objetos. O editor manipula um dado diagrama como um conjunto de páginas de tamanho fixo. As páginas podem conter **caixas** (representam classes/objetos) e **ligações** (representam as associações entre as classes) e **textos** (representam o nome de uma classe, nome de um atributo, nome de uma operação ou nome de uma associação).

Apesar da limitação da ferramenta em editar apenas Diagramas de Objetos, os autores comentam que a extensão para diagramas dos outros modelos (diagrama de transição de estados e diagrama de fluxo de dados) pode ser realizada, porém não fornecem maiores esclarecimentos sobre esta questão. A experiência adquirida no emprego do método OMT durante o desenvolvimento da ferramenta é relatada em termos de vantagens e desvantagens observadas.

2.2.5 Edge [PT90]

A funcionalidade da ferramenta Edge permite:

- mostrar o grafo usando um de quatro algoritmos para desenho automático: *Sugiyama* [STT81] e *Robins* [Rob87] para grafos dirigidos, *Reingold* [RT81] para árvores e *Woods* [Woo81] para grafos planares não dirigidos.
- editar o grafo com operações de adição, remoção e alteração de seus nós e arestas.
- criar subgrafos e associá-los a diferentes níveis de abstração.

Como grande parte de diagramas apresenta uma estrutura em grafo, a ferramenta Edge poderia ser uma forma mais abstrata para manipular diagramas, além de possuir uma importante função: desenho automático. Entretanto, os algoritmos consideram apenas nós de tamanho fixo e isto não ocorre com muita freqüência, conforme observado no levantamento de diagramas relatado anteriormente.

2.3 Comentários

As principais referências para a Seção 2.1, sobre diagramas, são [MM85, Tri88, Dav88]. Nelas encontram-se descritas várias representações gráficas, que permitem

descrever modelos típicos da área de computação. A Seção (2.2) apresenta os trabalhos correlatos, dentre eles os mais importantes são os desenvolvidos pelo grupo de pesquisa do ICMSC/USP (veja relação de trabalhos na Seção 1.5) e o Unidraw [VL90].

Capítulo 3

Métodos Orientados a Objetos e o Método OMT

Neste capítulo são apresentados alguns métodos Orientados a Objeto (OO) e, em especial, o OMT. A Seção 3.1 relata as razões pelas quais o paradigma de objetos e este particular método foram escolhidos. A realização de um estudo comparativo dos principais métodos OO em uso é uma tarefa não trivial. No presente trabalho, optou-se em compilar publicações específicas sobre esse assunto e utilizá-las na escolha de um dos métodos considerados. A Seção 3.2 descreve sucintamente o processo de modelagem conforme o método OMT, bem como os modelos e as notações empregadas.

3.1 Métodos Orientados a Objetos

O objetivo desta seção é justificar a escolha do método OMT, dentre os vários métodos orientados a objetos, para a modelagem da plataforma.

Características do processo de desenvolvimento OO:

- *Dedicação maior do tempo de desenvolvimento nas fases iniciais.* Grande parte do tempo de todo o desenvolvimento é dedicado às fases de Análise e Projeto. Por outro lado, a atividade de implementação tende a ser mais rápida e simplificada, pois muitas das decisões já foram tomadas nas fases anteriores.

- *Ênfase no domínio da aplicação.* Um sistema desenvolvido com base na sua funcionalidade é muito mais suscetível a grandes mudanças e, dependendo da

situação, podem ser muito extensas as conseqüências de uma alteração. Nos sistemas desenvolvidos de acordo com o paradigma de objetos, a ênfase é em objetos (estrutura de dados + comportamento), cuja estrutura é mais estável (menos sujeitos a mudanças) e modular. Qualquer que seja a fase de desenvolvimento, o conceito de objeto é o mesmo, diferentemente do que ocorre com funções.

- *A fronteira entre as fases não é precisamente delimitada.* Pelo fato mencionado no item anterior – o conceito de objeto é o mesmo em qualquer fase – a diferença está nos tipos de detalhes que são enfocados em cada fase. Esta característica não é exclusiva do desenvolvimento OO.

Para destacar a importância do paradigma de objetos neste trabalho, novamente serão mencionadas as propriedades desejáveis à plataforma:

- Extensível: com o intuito de facilitar a realização de modificações e extensões;
- Reutilizável: a fim de que as suas características possam ser reaproveitadas em novas definições ou evitar redundâncias desnecessárias:

Na tentativa de satisfazer estas propriedades, o paradigma de objetos tem se mostrado bastante adequado, principalmente pela ênfase que dá em conceitos como: **encapsulamento**, separação dos aspectos do objeto visíveis externamente e acessíveis por outros objetos, dos detalhes de implementação, que ficam ocultos. Assim sua implementação pode ser alterada (para melhorar o desempenho ou corrigir um defeito, por exemplo) sem afetar os objetos da aplicação que usam seus serviços; **herança**, mecanismo que permite o compartilhamento de características entre classes, preservando as diferenças; e **abstração**, mecanismo que ajuda na manipulação e entendimento de conceitos complexos, considerando apenas os detalhes relevantes em um determinado contexto.

Em [RBP+91], Rumbaugh **et al.** tentam caracterizar o que é orientação a objetos através de quatro aspectos que devem estar presentes: **identidade**, significa que um objeto é distinto do outro, mesmo possuindo atributos com os mesmos valores; **classificação**, objetos com o mesmo conjunto de atributos e mesmo comportamento podem compartilhar uma descrição que recebe a denominação de *classe*; **polimorfismo**, operações sintaticamente semelhantes porém com semântica diferente, pois dependem da classe onde foram definidas; e **herança**, mecanismo que permite a classes compartilharem atributos e operações com outras classes.

3.1.1 Exemplos de Métodos

Atualmente existem vários métodos OO em uso e muitas opiniões sobre qual a melhor ou a mais recomendada. Alguns exemplos de métodos são apresentados sucintamente a seguir. Como o interesse deste trabalho não é fazer uma avaliação minuciosa dos métodos, foi mais conveniente utilizar resultados já existentes.

De um modo geral, os processos de desenvolvimento baseados em métodos OO são bastante semelhantes entre si. A partir dos requisitos ou da descrição do problema são definidos os objetos da aplicação, os atributos e métodos das classes, as relações entre as classes (estrutura estática), as interações entre eles (controle) e as funções responsáveis pelas transformações nos objetos (funcionalidade). As diferenças mais evidentes referem-se à fase do processo de desenvolvimento em que cada atividade deve ser realizada, as formas de representação das informações, a terminologia usada e o procedimento para realização das atividades.

Booch [Boo91]

- baseia-se no modelo espiral;
- dá mais ênfase às fases de Projeto e Implementação;
- recomenda a construção de protótipos;
- os modelos são representados graficamente através de:
  - Diagramas Estáticos: representa classes, objetos, módulos e processos;
  - Diagramas Dinâmicos: representa as transições de estados, "timing" e a comunicação entre objetos derivados de classes;
  - Templates: mecanismo de documentação.

Coad & Yourdon [CY91]

- dá mais ênfase à fase de Análise e orienta sobre como conduzir as fases de Projeto e Implementação;
- possui um conjunto de heurísticas úteis na identificação das classes;
- a comunicação entre classes (troca de mensagens) baseia-se no modelo cliente-servidor;

- utiliza terminologias não muito usuais como, por exemplo, *assunto* e *serviço*;
- as representações gráficas utilizadas são:
  - Diagrama de Objetos e Diagrama de Classes: extensão do Modelo Entidade Relacionamento. Consiste de 5 camadas, cada uma para um nível diferente de detalhe (classes e objetos; estruturas; assuntos; atributos; e serviços);
  - Diagrama de Estados: um diagrama simples que mostra todos os possíveis estados de um objeto e as transições permitidas entre estados;
  - Diagrama de Serviços: auxilia na representação do modelo cliente-servidor.

Rumbaugh et al. [RBP+91] (OMT - Object Modeling Technique)

- dá mais ênfase à fase de Análise e muito pouca ênfase à Implementação. Cobre todas as fases de desenvolvimento;
- as representações gráficas são variações (extensões) de representações bastante difundidas, tais como Diagrama Entidade Relacionamento, Diagrama de Transição de Estados e Diagrama de Fluxo de Dados;
- representa uma mudança "incremental" com relação aos métodos tradicionais [FK92], ou seja, o impacto é atenuado, principalmente pelo que consta no item anterior;
- as representações gráficas utilizadas são:
  - Diagrama de Objetos: representa a estrutura estática dos objetos;
  - Diagrama de Transição de Estados (*Statecharts*): representa os estados dos objetos, transições de estados e eventuais interações entre objetos;
  - Diagrama de Fluxo de Dados: representa as transformações às quais objetos e seus atributos estão sujeitos;
  - Diagrama de Arquitetura do Sistema: captura o relacionamento estático entre os subsistemas.

Obs: Mais informações sobre este método são dadas na Seção 3.2.

Wirfs-Brock et al. [WBWW90]

- dá mais ênfase às fases de Projeto e Implementação;
- baseado no modelo cliente-servidor, ou seja, o sistema é composto por uma coleção de servidores, com determinadas responsabilidades, e que também prestam serviços aos clientes, baseados nos contratos que definem a natureza e escopo das interações;
- é mais apropriado ao desenvolvimento de pequenos projetos;
- introduz terminologias pouco conhecidas como *contrato*, *responsabilidades* e *colaborações*;
- as representações gráficas utilizadas são:
  - Diagrama de Hierarquia de Classes: mostra as hierarquias de classes relacionadas por herança;
  - Diagrama de Venn: mostra as sobreposições de responsabilidades entre classes, o que ajuda na identificação de potenciais superclasses abstratas;
  - Diagrama de Colaborações: mostra as classes, subsistemas, contratos e colaborações;
  - Cartões de Contrato: documentam os contratos entre clientes e servidores.

3.1.2 Estudos Comparativos

Monarchi & Puhr [MP92]

Monarchi e Puhr avaliam métodos de Análise e Projeto Orientado a Objetos, identificando componentes críticos e usando-os como base para comparações. Também são feitas algumas observações acerca das áreas de pesquisa mais fortes e mais fracas, no sentido de orientar nichos de atuação para trabalhos futuros. O interesse por este artigo foi devido aos resultados das comparações que é relatado.

Os autores analisaram os seguintes métodos: Alabiso, Bailin, Booch, Coad Yourdon, Livari, Kappel, Lee Carver, Lieberherr et al., Meyer, Rumbaugh et al., Shlaer Mellor, Wirfs-Brock et al., Ackroyd Daum, Beck Cunningham, Cunningham Beck, Page-Jones et al., Wasserman et al., Wilson, Bulman, Henderson-Sellers Constantine, Johnson Foote e Scharenberg Dunsmore. São ao todo 23 métodos classificados em três categorias: processos somente (Bulman, ..., Scharenberg Dunsmore); representações

somente (Ackroyd Daum, ..., Wilson); e processos e representações simultaneamente (as demais).

A primeira categoria, a de processos, refere-se a métodos para a realização da Análise e Projeto orientado a objetos. Esses processos não consideram os diagramas ou notações empregados. A segunda categoria, a de representações, refere-se a notações gráficas ou diagramas para descrição das saídas da Análise ou Projeto. O enfoque está na visualização das representações e não como são derivadas (não há técnicas associadas). A terceira categoria, a de processos e representações, abrange tanto os processos para realização da Análise e Projeto quanto as representações para registro dos resultados.

Os métodos coletados foram publicadas entre 1988 e 1991. Embora existam outras publicações antes deste período, o enfoque era dado principalmente à programação orientada a objetos ou conceitos gerais sobre este paradigma. Mais importante do que esta classificação é a relação de componentes críticos, os quais também são divididos em categorias: Recursos para a Análise do Domínio do problema (identificação da estrutura e comportamento); Recursos para o projeto da solução, com enfoque na interface, aplicação e utilitários (identificação da estrutura e comportamento); Recursos para a representação das informações estruturais e comportamentais, inclusive restrições; e Recursos para o gerenciamento da complexidade estrutural e comportamental de grandes sistema (mecanismos de particionamento do problema e divisão em subsistemas).

Dentre os 28 componentes críticos identificados, apenas 3 métodos obtiveram resultados muito bons. São eles: Booch (21 componentes presentes), Rumbaugh et al. (20 componentes presentes) e Coad Yourdon (19 componentes presentes).

Em resumo, esta avaliação considera sobretudo a presença de recursos importantes na Análise e Projeto orientados a objetos. Os três métodos mais bem sucedidos na avaliação são, portanto, os mais completos.

Walker [Wal92]

Walker apresenta uma lista de características que um método de Projeto Orientado a Objetos deve possuir para merecer tal designação. Do mesmo modo que Monarchi e Puhr em [MP92], esta lista é usada posteriormente para avaliar métodos convencionais e orientadas a objetos. Os critérios da avaliação são baseados na verificação da presença ou não das características desejadas. Dentre os métodos avaliados estão JSD[1] (Jackson Structured Design), Alabiso, Pun Winder, Wasserman et al. (OOSD

[1]JSD não é considerado um método orientado a objetos, contudo, ele é incluído na comparação por causa do conceito *Long Running Process*, que pode ser útil na identificação de operações das

– Object-Oriented Structured Design), HOOD (Hierarchical Object Oriented Design), Booch, e Rumbaugh et al.

As características identificadas são divididas em 4 categorias: capacidade de abstração (suporte a agregação, hierarquia e generalização, métodos para identificação de estados, serviços e interfaces), expressividade da notação (representações para agregação, hierarquias, relações, troca de mensagens, restrições e assim por diante), suporte à validação da especificação (regras para transformação das representações, suporte à geração de casos de teste) e capacidade de reutilização (heurísticas para identificação de classes abstratas).

Dentre as 16 características identificadas, apenas 2 métodos se saíram muito bem. São eles: Booch e OMT, ambas com 15 características satisfeitas.

Fichman & Kemerer [FK92]

Diferentemente dos trabalhos comparativos anteriores, este artigo apresenta uma comparação entre métodos convencionais (MCs) e orientados a objetos (MOOs). O artigo descreve sucintamente as principais características e contribuições de vários métodos estruturados, tais como Yourdon Constantine, DeMarco, e Yourdon. Destaca também o surgimento dos métodos orientados a dados no início dos anos 80. E, completando a coleção, os autores discutem sobre os métodos orientados a objetos com enfoque na Análise (OOA), Projeto (OOD) e em ambas as fases (OOAD). O papel da OOA é, basicamente, o desenvolvimento de uma representação do problema. Exemplos de métodos OOA incluem: Coad Yourdon, Bailin, e Shlaer Mellor. O Projeto orientado a objetos, por sua vez, mapeia os requisitos do sistema para uma representação abstrata de implementação, de acordo com as restrições de Projeto. Os métodos OOD selecionados por Fichman e Kemerer são: Booch, Wasserman et al., e Wirfs-Brock et al. Somente dois métodos OOAD são mencionados: Coad Yourdon e Rumbaugh et al.

Nota-se que o emprego da tecnologia de Orientação a Objetos é cada vez mais crescente. Diante deste quadro, é importante conhecer quais os reais benefícios proporcionados. Este artigo foi importante, pois abre uma discussão sobre áreas que ainda são deficitárias e levanta questões que evidenciam as contribuições introduzidas pelos MOOs.

Com relação às comparações, elas enfocaram as fases de Análise e Projeto. Para a Análise, a comparação baseou-se na presença de 11 componentes, também chamados de dimensões de modelagem. Para o Projeto, foram identificados 10 componentes.

quais objetos participam.

A escolha destes componentes foi feita de modo que os componentes da Análise pudessem ser mapeados para um componente similar da fase de Projeto. Como resultado, verificou-se que muitos dos componentes da fase de Projeto não estavam presentes nas MCs. A interpretação deste fato é a seguinte: na transição entre as fases, o *gap* entre os modelos da Análise e do Projeto é muito grande. Nos MOOs esta transição é bastante "suave," muitas vezes esta fronteira é até obscura. Esta é uma vantagem enfatizada dos MOOs.

Outra vantagem atribuída à orientação a objetos é a questão de reutilização. Quando apropriadamente aplicada, os mecanismos como encapsulamento, herança e polimorfismo têm proporcionado benefícios. Muitos MOOs se concentram em orientar no sentido de promover a reutilização e se esquecem de abordar o outro extremo, ou seja, o seu usufruto. Pelo menos dois métodos têm tido esta preocupação: Coad Yourdon e Booch. Elas destacam a necessidade de se examinar componentes prévios e, se possível, acoplá-los em novas aplicações. Entretanto, os métodos não dizem como encontrar e avaliar tais componentes para reutilização. Este assunto é tratado em mais detalhes em [CB91], onde os autores sugerem que as organizações tenham pessoas especializadas em reutilização e segregadas da equipe de desenvolvimento. Uma discussão mais pessimista, ou realista, sobre orientação a objetos ("fantasias" e áreas problemáticas) pode ser encontrada em [BE94].

Um problema no uso de diferentes métodos para Análise e Projeto é a combinação de terminologias e notações que não sejam compatíveis. Métodos de Análise como Coad Yourdon e Shlaer Mellor antecipam muitas das responsabilidades da fase de Projeto. O oposto também ocorre com Wirfs-Brock. Por isso, na escolha do método a ser utilizado no desenvolvimento da plataforma foi dada uma preferência a métodos que contemplassem todas as fases.

As principais diferenças entre os MCs e os MOOs residem nos conceitos próprios da orientação a objetos, tais como classe, herança, método e protocolo de mensagens. O conceito de encapsulamento pretende coibir a violação de muitos dos tipos de acesso praticados nos MCs. Por exemplo, a decomposição funcional permite acessos a qualquer entidade. Isto não é desejável segundo o paradigma de objetos. Outro conceito com uma interpretação diferente é o de modularidade: em sistemas convencionais este conceito está relacionado com programas, sub-rotinas e funções; em sistemas orientados a objetos, o próprio objeto é uma unidade primária de modularidade. Estas diferenças são algumas das razões pelas quais muitos profissionais de Computação ainda receiam mudar para o paradigma de objetos. Este receio não é justificável, pois muitos dos conceitos de orientação a objetos têm origem em experiências bem sucedidas do passado – modularidade, abstração, encapsulamento, reutilização – e possuem uma base teórica bem fundamentada.

Apesar dos MOOs serem recentes, portanto com um nível de maturidade menor do que os métodos convencionais, as evoluções introduzidas a medida que as pesquisas e as experiências são acumuladas têm trazido muitas contribuições. Muitas ferramentas de suporte já se encontram disponíveis comercialmente, tais como ObjectTeam/Cadre e StP/IDE.

Karam & Casselman [KC93]

Karam e Casselman definiram um procedimento de avaliação e comparação de métodos para desenvolvimento de sistemas (SDM - Software Development Method), que serve inclusive para métodos orientados a objetos.

De acordo com os autores, um SDM de sucesso deve:

- dar suporte à síntese de produtos;
- possibilitar a análise de produtos;
- melhorar a comunicação entre os membros de uma equipe de desenvolvimento;
- agir como uma forma de gerenciador ou coordenador do processo de desenvolvimento e
- facilitar a manutenção e evolução do produto de software resultante.

Como requisitos para as comparações, foi estabelecido que o procedimento de avaliação e comparação deve ser: **geral**, para que seja aplicável aos mais diversos tipos de métodos; **preciso**, para que as semelhanças e diferenças entre os métodos possam ser identificadas; **compreensível**, para que os principais aspectos dos métodos sejam destacados; e **balanceado**, para que os aspectos técnicos, gerenciais e de uso sejam considerados adequadamente.

O procedimento de comparação é composto por uma lista de 21 propriedades, cujos valores recebidos são numéricos ou narrativos. Valores numéricos são mapeados para os níveis A, B ou C, onde o nível C equivale ao valor que mais se aproxima do objetivo da propriedade.

Nos exemplos de aplicação do procedimento são usadas dois métodos: Rumbaugh et al. e Buhr.

- O conteúdo desse artigo não teve nenhuma contribuição significativa ao processo de escolha do método, que está em discussão. Entretanto, ele é apresentado pela importância do seu tema. Em uma avaliação mais minuciosa, este procedimento poderia servir de subsídio na escolha de um método.

3.1.3 Escolha do Método OMT

A escolha do método a ser seguido no presente trabalho deu-se em função das necessidades do próprio projeto, das comparações apresentadas na seção anterior e dos recursos presentes no método OMT. Os principais fatores que influenciaram nesta decisão foram:

- segue fundamentos do paradigma de objetos, um recurso importante na busca de propriedades como extensibilidade e reusabilidade para a plataforma;
- suporta todas as fases de desenvolvimento, mantendo os mesmos conceitos e notações durante todo o processo.
- usa conceitos e notações bastante difundidas. Isto facilita o seu aprendizado. A notação de Booch é considerada mais rica [BBJS92], porém não é tão simples de assimilar;
- obteve bons resultados nas comparações entre métodos, conforme a Seção 3.1.2.

3.2 Método OMT

Esta seção descreve sucintamente o método OMT, cujo objetivo é ajudar no entendimento do processo de desenvolvimento da plataforma, assunto do Capítulo 4.

O processo de desenvolvimento, segundo o método OMT, consiste na construção e refinamento de modelos para o domínio da aplicação, sem mudança nos conceitos e notações utilizadas. As representações dos modelos geradas em cada fase também servem como documentação de projeto. O conceito fundamental é o **objeto**: entidade que incorpora informações estruturais e comportamentais. A seguir serão descritos os modelos utilizados.

3.2.1 Os Modelos

Modelo é uma forma abstrata de abordar um problema antes de se iniciar a implementação da solução. Os detalhes não essenciais a um determinado aspecto do problema são ignorados, dependendo do nível de abstração enfocado. No método OMT, os modelos são empregados durante todas as etapas de desenvolvimento. Progressivamente mais detalhes são incorporados até que se obtenha algo bastante próximo da implementação. Para os modelos, as estruturas de dados (árvores, listas) usadas

no sistema são detalhes de mais baixo nível e, portanto, são consideradas apenas em etapas posteriores mais próximas da fase de Implementação. Os modelos do método OMT capturam três pontos de vista diferentes e complementares de um sistema: (1) Modelo de Objetos, descreve a estrutura estática do sistema e a organização dos dados em termos de objetos e seus relacionamentos; (2) Modelo Dinâmico, descreve as transições de estados de objetos de uma classe bem como as interações entre objetos; e (3) Modelo Funcional, descreve as transformações nos dados do sistema.

A notação gráfica utilizada para representação dos modelo é apresentada no decorrer do Capítulo 4. A seguir é feita uma introdução aos conceitos empregados em cada modelos.

Modelo de Objetos

O Modelo de Objetos (MO) é o mais importante dentre os três mencionados. Representa a estrutura estática dos objetos no sistema e seus relacionamentos. Serve de base para a definição dos outros modelos. Os conceitos usados são:

objeto: é um conceito, abstração, ou algo com uma fronteira bem definida, cujo significado, para o problema, promove o entendimento do mundo real e fornece a base para a implementação.

classe: descreve uma categoria de objetos com a mesma estrutura de dados (atributos) e mesmo comportamento (operações). O conjunto de atributos e operações é denominado de **característica** da classe/objeto. A instância de uma classe é chamada de objeto;

atributo: propriedade de objeto descrita na respectiva classe;

operação: ação que um objeto realiza ou a que está sujeito. Costuma ser chamada de método, quando implementada em alguma linguagem de programação;

associação: descreve uma relação entre objetos. A instância de uma associação, isto é, a associação entre objetos específicos é denominada de **link**;

especialização/generalização: relacionamento entre uma classe (superclasse) e seus refinamentos ou especializações (subclasses);

herança: mecanismo para compartilhamento/fatoração de atributos e operações usando o relacionamento de especialização/generalização.

O modelo é representado graficamente como uma extensão do modelo entidade relacionamento que combina conceitos de orientação a objetos (classe e herança) e modelagem de informação (entidades e relações).

Modelo Dinâmico

O Modelo Dinâmico (MD) descreve os aspectos de um sistema que mudam ao longo do tempo. O processo de sua construção consiste em definir as possíveis seqüências de estados dos objetos e as atividades que devem ser executadas em resposta aos estímulos internos e externos. As informações sobre a implementação das atividades não são relevantes. Os conceitos do modelo são os seguintes:

estado: valores assumidos pelos atributos do objeto em um determinado instante de tempo;

evento: estímulo enviado de um objeto a outro. Em resposta a um evento pode, eventualmente, ocorrer uma mudança de estado em determinados objetos;

transição: mudança do estado de um objeto causada pela ocorrência de um evento específico;

ação: operação executada "instantaneamente," durante uma transição de estado de um objeto;

atividade: operação executada durante a permanência de um objeto em um determinado estado.

Esses conceitos são representados através de uma variante de um tipo de Diagrama de Transição de Estados (DTE), chamada statecharts. O diagrama é uma rede de estados organizados de forma hierárquica, do mesmo modo que o diagrama de objetos é uma rede de classes e relacionamentos. Para cada classe com um comportamento não trivial deve-se definir um DTE. A junção dos vários DTEs constitui o Modelo Dinâmico.

Modelo Funcional

Complementando os modelos anteriores, o Modelo Funcional (MF) descreve o aspecto do sistema que envolve a transformação dos dados. O modelo captura o que o sistema faz sem a preocupação de como e quando as transformações são efetuadas. Os conceitos presentes no modelo são:

processo: realiza a transformação nos dados do sistema;

fluxo de dados: indica os dados que fluem entre processos;

entidade externa: objeto que produz ou consome dados do sistema;

depósito de dados: objeto passivo que armazena os dados para consultas posteriores, não realiza nenhum tipo de processamento.

A notação gráfica empregada é a de Diagramas de Fluxo de Dados (DFDs). Um DFD é um grafo que mostra os processos envolvidos nas transformações de dados de entrada para geração de saídas. A seqüência de execução dos processos bem como decisões sobre os caminhos a seguir são responsabilidades do MD. Os vários DFDs resultantes expressam os dados requeridos por operações definidas no MO e por ações e atividades no MD. A coleção dos DFDs constitui o Modelo Funcional.

Relacionamento entre os modelos

Os três modelos do método OMT capturam aspectos distintos e complementares de um sistema. Cada modelo possui informações que precisam estar consistentes com informações presentes nos demais modelos.

Na Figura 3.1 encontra-se uma ilustração do relacionamento entre os modelos. A seqüência de leitura da figura é no sentido anti-horário, começando pelo Modelo de Objetos. Durante o processo de modelagem do método OMT, definem-se principalmente as classes e relações entre elas (associação, agregação, herança). Em seguida constrói-se o Modelo Dinâmico referente às classes com um comportamento dinâmico mais complexo. Alguns dos elementos definidos no Modelo Dinâmico referentes a essas classes são as ações e as atividades, às quais correspondem a operações que, via de regra, são incluídas no Modelo de Objetos. Por fim, de posse do Modelo Objetos atualizado ou das ações e atividades do Modelo Dinâmico, são construídos os DFDs correspondentes às operações mais complexas. Do Modelo Funcional obtido surgem novas operações (bem como parâmetros de entrada e saída) a serem integradas à descrições de classes do Modelo de Objetos. Estas são algumas das correspondências entre elementos dos modelos. Em [RBP+91] há mais informações sobre estes relacionamentos, porém não são muito claramente expostas.

3.2.2 Fases de Desenvolvimento

No livro de Rumbaugh et al., sobre o método OMT, estão descritos os procedimentos de aplicação do método nas fases de Análise, Projeto e Implementação. As atenções

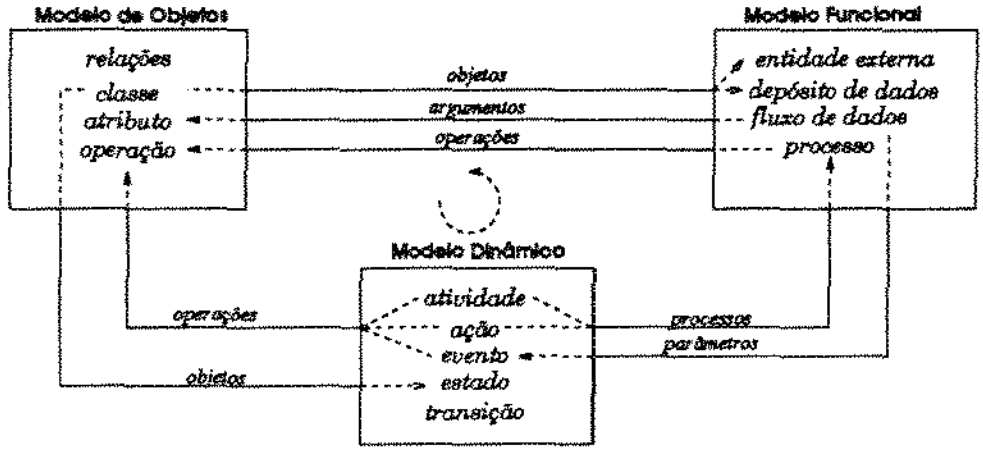


Figura 3.1: Relação entre os três modelos.

do livro concentram-se mais nas duas primeiras fases. A parte de implementação é discutida usando-se casos específicos de aplicações para linguagens de programação orientadas a objetos, linguagens não orientadas a objetos e sistemas gerenciadores de banco de dados. Segundo os autores, a tarefa de implementação deveria ser algo quase que mecânico, pois todas as principais decisões já foram tomadas.

A Figura 3.2 sintetiza a aplicação do método durante as fases de Análise e Projeto. Consiste em capturar as informações dos dois extremos (domínio do problema e domínio da implementação) e gerar os três modelos. Como se pode ver, o Modelo de Objetos foi colocado em destaque pois ele é o modelo principal, enquanto os demais são complementares à modelagem. Conforme o tipo de aplicação, a importância dos outros dois modelos pode ser maior ou menor. Por exemplo, o Modelo Dinâmico é imprescindível em sistemas com uma grande interação com o usuário.

A seguir, é feita uma descrição resumida das atividades envolvidas nas fases de Análise e Projeto do método OMT.

Análise

O objetivo da Análise é entender e registrar o que o sistema terá que fazer e não como será feito. Para tal, deve-se examinar a descrição informal do problema, analisar suas implicações e redefinir suas características em um formato mais rigoroso. Apenas as características relevantes ao "mundo real" devem ser consideradas nessa primeira fase. Os passos para a execução da fase de Análise são os seguintes:

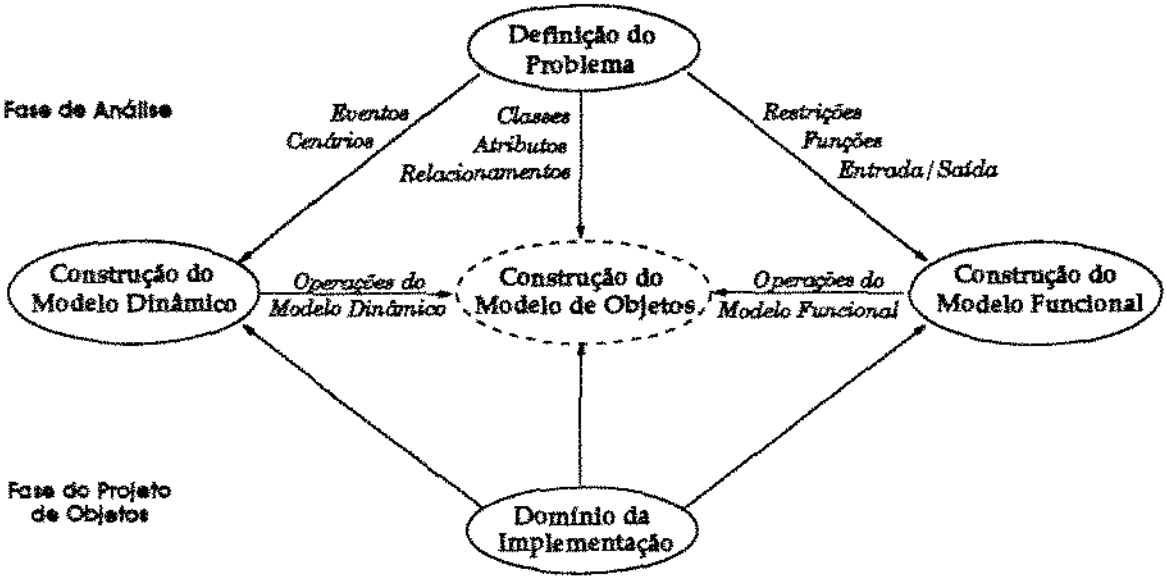


Figura 3.2: Síntese do processo de modelagem do método OMT [BBJS92].

1. Escrever ou obter[2] uma descrição informal do problema (Definição do Problema);
2. Construir o Modelo de Objetos:
   - Identificar classes;
   - Preparar um dicionário de dados contendo a descrição de classes, atributos e associações;
   - Identificar associações entre classes;
   - Adicionar atributos e *links* a classes;
   - Organizar e simplificar classes usando o mecanismo de herança;
   - Testar possíveis caminhos de acesso usando cenários;
   - Agrupar classes em módulos.

   Modelo de Objetos = diagrama de objetos + dicionário de dados.
3. Construir o Modelo Dinâmico:
   - Preparar cenários para seqüências de interação;

[2] O método OMT não fornece uma orientação mais detalhada sobre como a definição do problema pode ser obtida.

- Identificar eventos entre objetos e preparar diagramas de rastreamento de eventos (*event-traces*) para cada cenário construído;
- preparar um diagrama de fluxo global de eventos;
- Desenvolver um diagrama de estados para cada classe cujo comportamento dinâmico seja importante;
- Verificar a consistência e a completitude dos eventos compartilhados entre os diagramas de estado.

Modelo Dinâmico = diagrama de estados + diagrama global de fluxo de eventos.

4. Construir o Modelo Funcional:

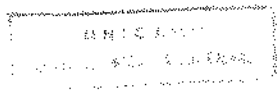
   - Identificar dados de entrada e saída;
   - Usar um diagrama de fluxo de dados para representar as dependências funcionais;
   - Descrever o que cada função deve fazer;
   - Identificar restrições;
   - Especificar critérios de otimização.

   Modelo Funcional = DFDs + restrições

5. Verificar e refinar os modelos:

   - Incorporar as operações que foram identificadas durante a preparação do Modelo Funcional no Modelo de Objetos;
   - Verificar consistências das classes, associações, atributos e operações. Comparar os três modelos obtidos com a definição do problema;
   - Desenvolver cenários detalhados, incluindo condições de erro.

Documentos da Análise = Definição do Problema + Modelo de Objetos + Modelo Dinâmico + Modelo Funcional.



Projeto do Sistema

Nesta fase, o projetista deve tomar decisões sobre a estrutura do sistema em construção, dividindo-o em subsistemas, se necessário. Esta fase é incluída no método para que o processo de desenvolvimento seja mais completo. Os passos realizados nesta fase são:

1. Organizar o sistema em subsistemas;
2. Identificar tarefas concorrentes;
3. Alocar os subsistemas a processadores;
4. Escolher uma estratégia de implementação para os depósitos de dados (arquivos, banco de dados);
5. Identificar recursos globais (processadores, espaço em disco, impressoras) e determinar os mecanismos de controle para o seu acesso;
6. Escolher uma estratégia de implementação para o controle do sistema: orientado a procedimentos, eventos, regras, concorrentes ou outros;
7. Considerar as situações especiais (inicialização do sistema, finalização do sistema, ocorrências de erro);
8. Estabelecer prioridades para a fase de Implementação.

Projeto dos Objetos

Com base nos modelos da Análise, o projetista deve incluir detalhes, de acordo com a estratégia traçada na fase de Projeto do Sistema. Com exceção da linguagem de programação a ser usada, deve-se definir as estruturas de dados, algoritmos e outros detalhes de implementação. O passos desta fase são:

1. Obter operações para o Modelo de Objetos a partir dos outros modelos:
   - Identificar operações provenientes do Modelo Funcional;
   - Identificar operações provenientes do Modelo Dinâmico.
2. Projetar a implementação de operações:
   - Escolher algoritmos que minimizem o custo de implementação;

- Selecionar estruturas de dados apropriadas para os algoritmos escolhidos;
- Definir novas classes e operações auxiliares, se necessário.

3. Otimizar os caminhos de acesso aos dados:

   - Adicionar associações redundantes para minimizar o custo de acesso;
   - Armazenar dados derivados de expressões complexas se elas forem necessárias repetidas vezes;

4. Ajustar classes para facilitar o uso do mecanismo de herança;

5. Projetar a implementação de associações;

6. Empacotar classes e associações em módulos.

Comentários sobre o método

- o método OMT não dá orientações mais detalhadas sobre como obter a descrição do problema;
- a Análise é a fase melhor definida pelo método, porém ele tem recebido críticas por se concentrar demais nesta fase;
- o método não fornece orientações adequadas sobre como certas entidades devem ser modeladas, deixando dúvidas com relação à escolha por um atributo ou uma classe para representar uma certa característica;
- o modelo dinâmico carece de ferramentas que mapeiem as especificações em código;
- não há uma correspondência ou relação bem definida entre os três modelos. Isto dificulta a verificação de consistência e integração dos modelos;
- a interface do sistema modelado não é devidamente abordada;
- a subdivisão da fase de Projeto (Projeto do Sistema e Projeto dos Objetos) suaviza o *gap* entre a Análise e a Implementação;

3.3 Comentários

O livro de Rumbaugh et al. [RBP+91] descreve detalhadamente o método OMT, que foi usado durante o processo de desenvolvimento da plataforma. Comparações entre métodos orientados a objetos podem ser obtidas em [MP92, Wal92, KC93, FK92], além de fornecerem uma visão crítica dos métodos em uso atualmente. Outros métodos promissores, mas que foram observados apenas depois da escolha e utilização do método OMT são Fusion [CAB+94] e Objectory [Jac92].

Capítulo 4

Desenvolvimento da Plataforma

Este capítulo descreve o desenvolvimento da plataforma, conforme o método OMT. O objetivo é apresentar o modelo da plataforma em cada fase do processo, que abrange aspectos estruturais, comportamentais e funcionais da plataforma.

A organização deste capítulo segue a mesma seqüência do processo de desenvolvimento do método OMT. Inicialmente, na Seção 4.1, é feita uma Análise baseada nos requisitos levantados na definição da proposta deste trabalho (Seção 1.2). Os resultados gerados nesta fase são os modelos preliminares da plataforma. A próxima fase é o Projeto, cuja finalidade é completar e detalhar os modelos definidos na fase anterior. O Projeto é dividido em duas etapas: Projeto do Sistema e Projeto dos Objetos. No Projeto do Sistema (Seção 4.2) é definida a arquitetura e seus subsistemas e são tomadas decisões estratégicas, tais como: recursos necessários (memória, ocupação em disco, gerenciador da base de dados) e compromissos com eficiência, portabilidade, usabilidade, confiabilidade, custo. Estas decisões afetam fundamentalmente a etapa do Projeto dos Objetos (Seção 4.3).

Para tentar esclarecer melhor os conceitos empregados na plataforma, o próximo capítulo apresenta um exemplo prático de utilização da plataforma. O exemplo serviu também como um estudo de viabilidade do emprego da plataforma.

4.1 Análise

A atividade desta fase compreende a construção de modelos da aplicação a partir dos requisitos previamente estabelecidos. Não é necessário modelar completamente a aplicação, mesmo porque existem detalhes que não são relevantes neste nível de abstração. Eventuais omissões de informação podem ser reparadas em revisões futuras.

Tão importante quanto definir os modelos é a tarefa de deixá-los consistentes.

Geralmente, os requisitos são especificados em uma linguagem informal, sujeitas a imprecisões e inconsistências. Assim, a fase de Análise é muito importante no sentido de tentar eliminá-las para evitar que se propaguem para as fases seguintes e comprometam todo o trabalho.

4.1.1 Requisitos da Plataforma

Os requisitos definem o que a plataforma deve fazer: *apoiar a construção de Ferramentas Baseadas em Diagramas (FBDs).* Na Figura 4.1 é apresentada uma arquitetura simplificada dos elementos principais decorrentes de um uso específico da plataforma. Uma particular FBD permite um determinado manuseio de diagramas de uma categoria específica denominada categoria alvo. Alguns exemplos de ferramentas e categorias de diagramas alvo estão indicados por setas.

Figura 4.1: Esquema que ilustra a relação plataforma-ferramenta-diagrama.

A plataforma deve ter predefinidas várias características básicas de FBDs. Desse modo, a principal tarefa é a de acrescentar as particularidades de cada caso, quando da construção de uma particular FBD. Para usar efetivamente o paradigma de objetos não basta simplesmente criar e manipular objetos. É preciso tê-los bem definidos. Por isso, pretende-se que a plataforma sirva como um modelo consistente e coerente na definição (especialização) das classes de uma FBD particular. Podem ser apontados, entre outros, os seguintes objetivos:

- de uma ferramenta: prover o acesso às funções de manipulação de diagramas da categoria alvo; fornecer um meio de visualização de diagramas dessa natureza;

- de uma categoria de diagramas: definir a organização das informações relativas aos componentes de diagramas e operações sobre esses diagramas; permitir a criação, remoção e alteração de componentes, armazenamento e busca de informações relevantes em um meio estável, como um banco de dados, bem como a verificação e manutenção da consistência de componentes e as relações entre eles.

É desejável que as informações gráficas de um diagrama sejam colocadas em um nível de abstração diferente das informações estruturais. O objetivo disto é dar liberdade a um diagrama de ser apresentado em diferentes formas e dispositivos gráficos. Com esta medida deve-se dar uma atenção especial à manutenção de consistência dos dois conjuntos de informações.

4.1.2 Análise

Nesta fase, os requisitos da plataforma são analisados e traduzidos para uma notação mais rigorosa e formal. As características básicas das FBDs são traduzidas em termos de três modelos: Modelo de Objetos, Modelo Dinâmico e Modelo Funcional.

Para que uma informação faça parte da plataforma é preciso analisar se ela é realmente genérica o suficiente. Esta tarefa já foi feita, em parte, no estudo das características básicas de diagramas de interesse.

O processo de construção dos modelos não segue obrigatoriamente uma seqüência uniforme de passos. Certos aspectos da aplicação só podem ser considerados depois que outros já foram definidos. Por isso, é mais adequado que os passos sejam realizados **iterativamente** dentro da mesma fase ou entre fases, neste tipo de desenvolvimento.

Serão apresentados a seguir os modelos resultantes desta fase. Como seria repetitivo descrever em detalhe o processo de construção dos modelos, considerando todas as iterações realizadas durante a fase de Análise, apenas o resultado final de cada passo é apresentado. Além disso, o método OMT utiliza os mesmos modelos em todo o processo de desenvolvimento. Por este motivo haveria muita redundância de informações.

Nos casos em que houveram mais de uma solução de modelagem, serão apresentadas as alternativas encontradas e uma explicação para a escolha de uma delas.

A notação dos modelos é relativamente simples. Isto se deve à semelhança com as notações bastante conhecidas das quais foi derivada. Uma descrição sucinta da notação utilizada pelo método OMT encontra-se no Apêndice A.

4.1.3 Modelo de Objetos

O Modelo de Objetos é recomendado como o ponto de partida da modelagem, pois ele não exige que se entre em muitos detalhes e está menos sujeito a alterações nos passos seguintes do processo de desenvolvimento.

Inicialmente, procurou-se identificar as classes significativas do ponto de vista do domínio do problema, as principais hierarquias de classes e, em seguida, as características a serem atribuídas a cada nível de abstração. Os resultados obtidos são descritos nos passos a seguir.

Identificação dos Objetos e Classes

Na fase de Análise é importante identificar as entidades (classes e relacionamentos) relevantes ao contexto de FBDs. Outras considerações fora deste contexto, mais próximas da implementação, devem ter a sua identificação adiada até a fase de Projeto.

No método adotado, uma classe só deve ser criada se ela possuir um significado dentro do contexto geral e isoladamente. Por exemplo, na plataforma não é preciso definir uma classe chamada **usuário**. Embora ele seja uma entidade que interage com os objetos da plataforma, suas informações isoladamente (por exemplo, nome, idade, endereço, profissão) não são relevantes. Em um sistema bancário, o usuário pode ser o cliente e, neste caso, sua existência como um objeto é justificada, pois obedece à condição anterior. Portanto, decidir se algo pode ser um objeto ou não depende essencialmente da aplicação.

As principais classes da plataforma são: **Diagrama** e **Ferramenta**. Com base nelas foram definidas as restantes.

No contexto de **Diagrama**, as classes relevantes são:

Diagrama: a maior parte de suas características já foi descrita na Seção 2.1.5. Possui informações relevantes ao diagrama como um todo, tal como os seus componentes.

Bloco: é um dos principais elementos de um diagrama. Dependendo do tipo de diagrama pode significar um dado ou uma atividade (processo, função, procedimento), conforme Figura 2.4.

Conector: juntamente com a classe **Bloco** constituem os principais elementos de um diagrama. Dependendo do tipo de diagrama e dos blocos interligados pode significar o relacionamento entre dados, um fluxo entre processos, uma seqüência de atividades, entre outros.

Representação Externa: representação do diagrama em um meio estável (arquivo, banco de dados). Em sistemas como o EDGE [PT90], o mapeamento para um arquivo é feito por funções geradas automaticamente, baseado na definição das estruturas usadas.

No contexto de **Ferramenta**, apenas a própria classe foi considerada relevante:

Ferramenta: responsável pela interação com o usuário na manipulação de diagramas. Captura a opção escolhida e solicita a execução da função correspondente. A função é constituída de chamadas a operações definidas em classes no contexto de diagramas. Uma ferramenta possui também, via de regra, uma região para desenho do diagrama.

Na relação entre as classes **Ferramenta** e **Diagrama** foram identificadas as classes:

LequeDeOpções: Coleção de opções;

Opção: Uma opção disponibilizada pela ferramenta ao usuário, que está associada a uma função, responsável por invocar operações, definidas em classes do contexto de diagramas.

No desenho de um diagrama está envolvida a manipulação de objetos gráficos que representam os elementos que o compõem. Os nomes adotados para classes estão em inglês. Muitas das bibliotecas gráficas encontradas possuem funções e atributos definidos em inglês e a tradução de alguns poderia gerar ambigüidades (por exemplo, *elbows*, *bitmap*, *right angles*, *angles*, *blob* – veja Figura 4.5):

Symbol: topo da hierarquia de símbolos gráficos usados para desenho de diagramas. Exemplos de símbolos são as figuras geométricas (retângulos, círculos, triângulos) e textos. Uma taxonomia de símbolos gráficos é ilustrada na Figura 4.5.

Uma classe chamada **Configuration** também foi definida. Esta classe deve conter os atributos de configuração para desenho dos símbolos gráficos, por exemplo, cor, tipo de letra, espessura da linha e outros.

Identificação das Associações

As associações expressam as relações entre as classes. A partir das classes identificadas no passo anterior, procura-se estabelecer as dependências entre elas a partir de frases como: um **diagrama** *consiste de* **blocos e conectores**; um **diagrama** *é mapeado para* uma **representação externa**; **blocos e conectores** são *representados graficamente* por **symbol**; **ferramenta** serve para *manipular* o **diagrama**. Na

Figura 4.2 é apresentada uma versão simplificada do diagrama de objetos da plataforma com as associações resultantes destas frases. A agregação `subOpções`, partindo e retornando em **LequeDeOpções**, permite que haja um relacionamento não linear no conjunto de opções, por exemplo, uma hierarquia de opções.

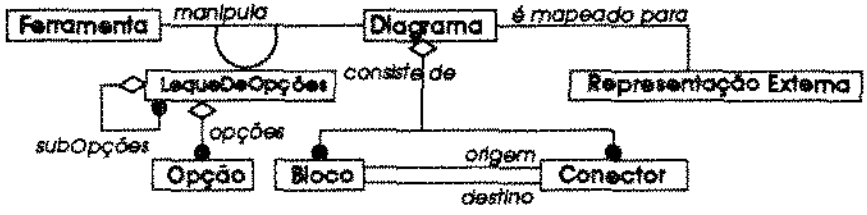


Figura 4.2: Classes e associações da plataforma.

Identificação dos Atributos

Os atributos são propriedades de objetos. Por exemplo, para uma pessoa seus atributos seriam possivelmente o nome, a cor, o peso e assim por diante. As classes e associações identificadas anteriormente são extraídas da Especificação de Requisitos da plataforma (Seção 4.1.1). No caso dos atributos isto também ocorre, porém foi preciso de algo mais, derivado de informações implícitas ao domínio da aplicação. Por isso, um estudo de diagramas foi fundamental.

A Figura 4.3 mostra o diagrama de objetos da figura anterior complementado com atributos. O atributo `operação`, pertencente à classe **Opção** corresponde a um ponteiro para uma função que retorna um resultado do tipo *Status* (Mais detalhes sobre o tipo *Status* são apresentados na Seção 4.1.6). Alguns atributos pertencentes ao domínio da implementação não foram incluídos. Cada linha de atributo no diagrama tem o seguinte formato: `nome_do_atributo: [tipo[=valor_default]]`. Quando a região dos atributos está vazia significa que a classe não possui atributos próprios.

Refinamento com Inclusão de Herança

Herança é um mecanismo que permite o compartilhamento de propriedades comuns entre classes. Entenda-se como propriedade o conjunto de atributos e operações de uma classe. Um outro fator muito importante para o compartilhamento é a semântica das classes envolvidas (conforme páginas 244, 284 e 285 de [RBP+91]). Pode ser induzido de duas maneiras: por **generalização** ou por **especialização**. Generalização é o processo de fatorar as propriedades comuns de várias classes, definindo-as em uma classe de nível hierárquico superior. Especialização é o processo de particularizar as

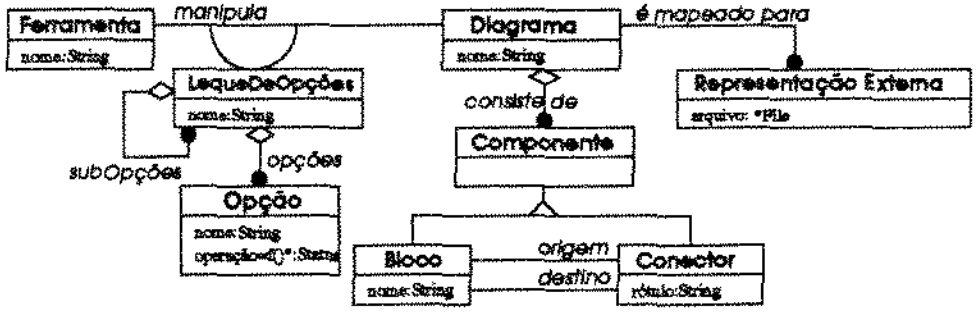


Figura 4.3: Diagrama de objetos da plataforma contendo classes, associações e atributos.

propriedades de uma classe, definindo novas características ou redefinindo as existentes. A classe que herda é chamada de **subclasse**, e a herdada de **superclasse**. A definição de classes pelo processo de generalização e especialização assemelha-se, respectivamente, aos processos de desenvolvimento de sistemas *bottom-up* e *top-down*.

O mecanismo de herança foi fundamental na definição da plataforma e será importante também na sua utilização. O processo de **generalização** foi usado na caracterização das classes e propriedades das FBDs. Através da observação de vários tipos de diagramas e ferramentas, procurou-se fatorar as características comuns entre eles, ou algo próximo a isto. Durante a utilização da plataforma, as características predefinidas são particularizadas pelo processo de **especialização**. Portanto, as duas formas de induzir herança são bastante importantes neste trabalho.

Até o momento, as classes **Bloco** e **Conector** foram tratadas separadamente. Através da generalização das propriedades destas classes foi criada a classe de nível mais genérico, chamada **Componente**. Como conseqüência, as duas associações de agregação *consiste-de*, entre **Diagrama** e **Bloco**, e entre **Diagrama** e **Conector** ficam reduzidas a apenas uma, entre **Diagrama** e **Componente**, como ilustra a Figura 4.4. Assim, o processo de generalização também pode ser empregado na redução do número de associações.

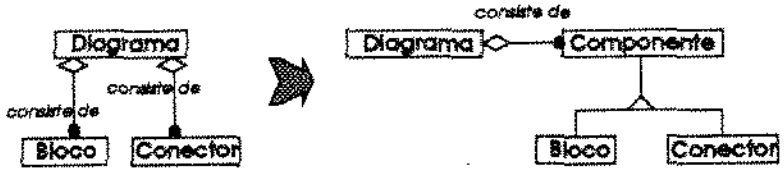


Figura 4.4: Generalização das classes bloco e conector.

Um exemplo de especialização é o da classe **Symbol**. Na Figura 4.5, **Symbol** é o topo da hierarquia de objetos gráficos. Nas subclasses de **Symbol** há vários tipos de

símbolos normalmente usados em diagramas. Caso um símbolo não esteja disponível na hierarquia, ele pode ser definido a partir do ramo mais apropriado da árvore, como um caso particular da superclasse em questão.

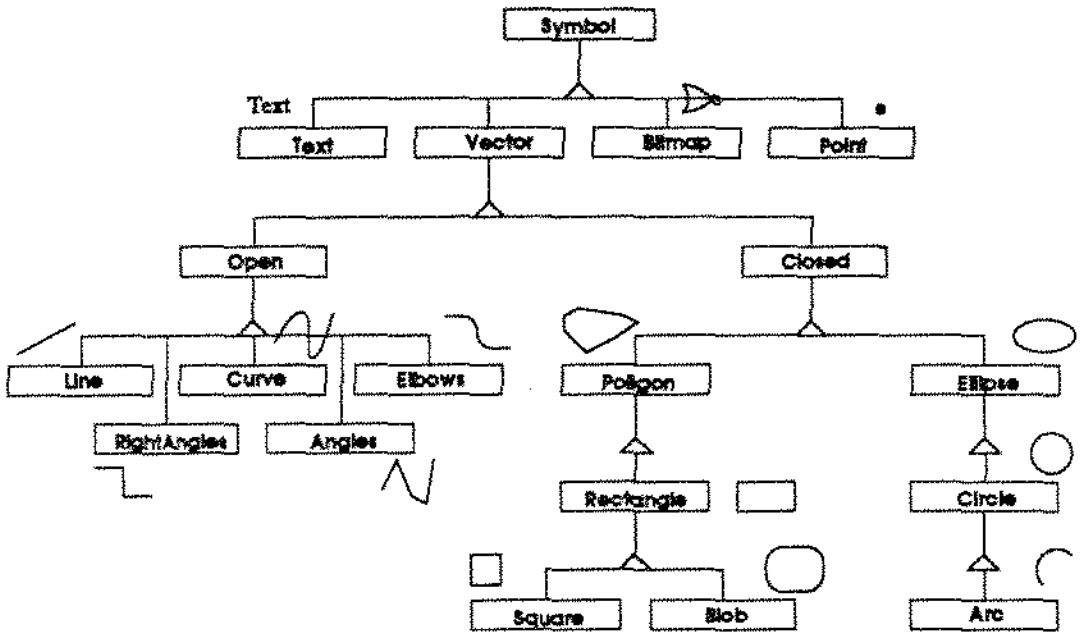


Figura 4.5: Hierarquia de objetos gráficos para desenho dos componentes de diagramas.

A hierarquia de objetos gráficos, ilustrada na Figura 4.5, foi baseada na taxonomia proposta por Barth em [Bar86]. Cada classe possui operações de criação, desenho, modificação e outras a serem citadas mais adiante. Um conceito implícito ao objeto gráfico é o de que a sua implementação em um ambiente gráfico específico deva ser transparente para as classes que requisitam os seus serviços (operações). Um mudança neste ambiente deve implicar em mudanças apenas na implementação dos serviços, mantendo o formato com que são requisitados, isto é, a mesma interface (parâmetros de entrada e saída dos serviços) deve ser mantida. Uma versão desta hierarquia na forma de classes, atributos e operações é apresentada na seção sobre a fase de Projeto dos Objetos (Seção 4.3.1).

Agrupamento das Classes em Módulos

Este passo da modelagem dos objetos é responsável pela segmentação das classes em agrupamentos, denominados módulos, de acordo com suas afinidades do ponto de vista semântico. Esta segmentação é recomendada em diagramas complexos, onde

a visualização e entendimento de todo o conjunto de classe simultaneamente pode ser muito complicada. Embora a quantidade de classes da plataforma não seja tão grande, uma subdivisão em módulos é útil, pois a tendência é o crescimento do número de classes criadas por especialização à medida que novas FBDs forem modeladas.

O Modelo de Objetos da plataforma foi fragmentado nos seguintes módulos:

1. *Diagrama* ← diagrama, componente e representação externa;
2. *Ferramenta* ← ferramenta;
3. *Gráficos* ← symbol e configuration.

O módulo *Diagrama* deve conter classes para a especificação dos diferentes tipos de diagramas, o módulo *Ferramenta* serve para a especificação dos diferentes tipos de ferramentas e o módulo *Gráficos* para a parte de apresentação dos diagramas. Classes como **Bloco** e **Conector** já estão contempladas na hierarquia originária da classe **Componente**. O relacionamento entre os módulos é determinado pelas associações entre classes de diferentes módulos. Para isto, basta consultar o Modelo de Objetos.

4.1.4 Modelo Dinâmico

O Modelo Dinâmico (MD) descreve o comportamento de objetos do sistema modelado em relação à dimensão tempo. Trata-se de seqüências de operações que são executadas em resposta a estímulos (**eventos**) trocados entre os objetos. A ocorrência dos eventos pode resultar em mudanças no **Estado** de objetos, representados graficamente através de diagramas de estado estendidos. O conjunto de diagramas resultante constitui o padrão de atividade do sistema ou o MD.

Apenas os comportamentos de objetos de classes que apresentam um comportamento mais complexo precisam ser descritos no modelo e compartilhados no nível de classe, isto é, cabem descrições de comportamento para classes cujos objetos têm uma interação intensa com outros objetos e/ou entidades externas (usuário, dispositivo periférico). Em aplicações como compiladores e folhas de pagamento, onde o comportamento é bastante simples, o MD pode até ser desnecessário.

O processo de modelagem dinâmica consiste em: identificar os eventos, através de cenários e *event-traces*; organizar as seqüências de estados em diagramas de estado; bem como identificar as possíveis inconsistências entre diagramas.

Na plataforma, apenas o comportamento genérico de objetos da classe **Ferramenta** será descrito neste modelo e, parcialmente, o da classe **Diagrama**. As outras

classes possuem comportamentos também importantes, mas que dependem muito mais das particularidades de cada FBD (por exemplo, **Componente** e **Symbol**) e devem ser definidos em suas especializações.

Serão descritos a seguir todos os passos intermediários da modelagem de comportamento associado à classe **Ferramenta**. Para a classe **Diagrama**, apenas o diagrama de estados resultante será apresentado.

Preparação de Cenários

Os cenários dão uma noção aproximada do comportamento do sistema. Neles são representadas as interações do sistema com o usuário, as informações trocadas e até os protótipos das telas em que os diálogos são realizados.

Normalmente, os requisitos da aplicação não revelam detalhes sobre as interações, ficando sob responsabilidade do analista decidir como as informações serão obtidas e quando serão produzidas. Em algumas aplicações esta tarefa é bastante complexa e de fundamental importância para o sucesso do produto. Por isto, existem pessoas com especialidade nesta atividade, chamadas de projetistas da interação [HH93].

Além das interações mais usuais, é preciso considerar casos especiais como entrada incorreta de dados, erros de processamento, interrupção de operações em execução e assim por diante. Segundo o método OMT, os casos usuais devem ser levantados primeiro. Cada cenário representa uma das possíveis seqüências de eventos. Ao juntá-los em um diagrama de estados, tem-se uma visão completa do comportamento de um objeto em particular, inclusive de caminhos indesejáveis e que não eram visíveis isoladamente. As informações trocadas através dos eventos são os seus atributos. Nos cenários, os atributos aparecem entre parênteses.

Veja a seguir os cenários da classe **Ferramenta**. O primeiro deles descreve uma seqüência normal.

Ferramenta solicita ao usuário a escolha de uma opção.
Usuário seleciona opção (escolha = verifica consistência).
Ferramenta executa a opção escolhida (escolha).
Execução é concluída (status = ok).
Ferramenta solicita ao usuário a escolha de uma opção.
Usuário seleciona opção (escolha = finaliza).
Ferramenta executa a opção escolhida (escolha).
Execução é concluída (status = finaliza).
Ferramenta finaliza atividades.

Cenários para situações não usuais (erro e cancelamento de execução):

Ferramenta executa opção escolhida (escolha).
Execução é concluída (status = msg_erro).
Ferramenta mostra mensagem ao usuário (status).
Ferramenta solicita ao usuário a escolha de uma opção.

Ferramenta executa opção escolhida (escolha).
Usuário cancela a execução.
Execução é concluída (status = msg_execução_cancelada).
Ferramenta mostra mensagem ao usuário (status).
Ferramenta solicita ao usuário a escolha de uma opção.

Formato da Interface

A Figura 4.6 ilustra um exemplo da janela de interação da ferramenta. O objetivo é ajudar na visualização das interações já definidas e na identificação de novas situações.

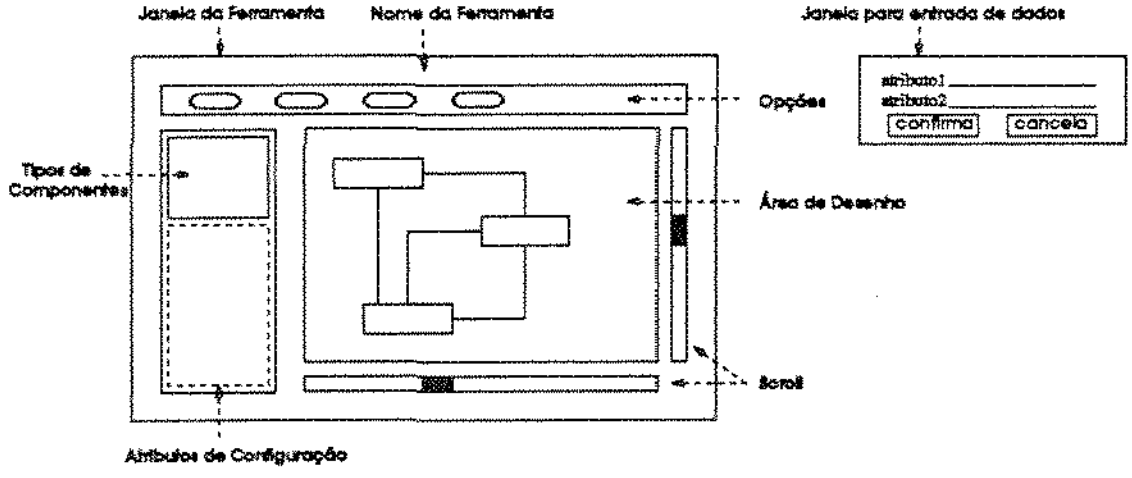


Figura 4.6: Formato de uma janela de interação da ferramenta com o usuário.

No topo da janela maior estão as opções de manipulação do diagrama, na lateral esquerda têm-se os atributos de configuração dos objetos gráficos e na lateral direita a região de desenho do diagrama. A região acima dos atributos de configuração contém os tipos de componentes relevantes ao diagrama, a escolha de um deles define por exemplo, qual o componente que se quer criar. A janela menor é necessária quando a

execução de alguma opção requer a entrada de dados do usuário. A opção `cancela` permite o cancelamento da operação executada, situação representada em um dos cenários anteriores.

Construção de *Event-Traces*

Para facilitar a visualização dos agentes envolvidos na emissão e recepção dos eventos, os cenários descritos anteriormente são mapeados para uma representação chamada *event-trace*. Todos os objetos participantes da interação são colocados em colunas de uma tabela e os eventos são representados como segmentos de reta interligando a coluna do objeto emissor com a do receptor. A Figura 4.7 mostra os *event-traces* correspondentes aos cenários da classe **Ferramenta**. Mesmo não sendo um objeto, o **usuário** é um agente participante da interação. Esta representação permite visualizar também os possíveis objetos que se relacionam com um objeto em particular.

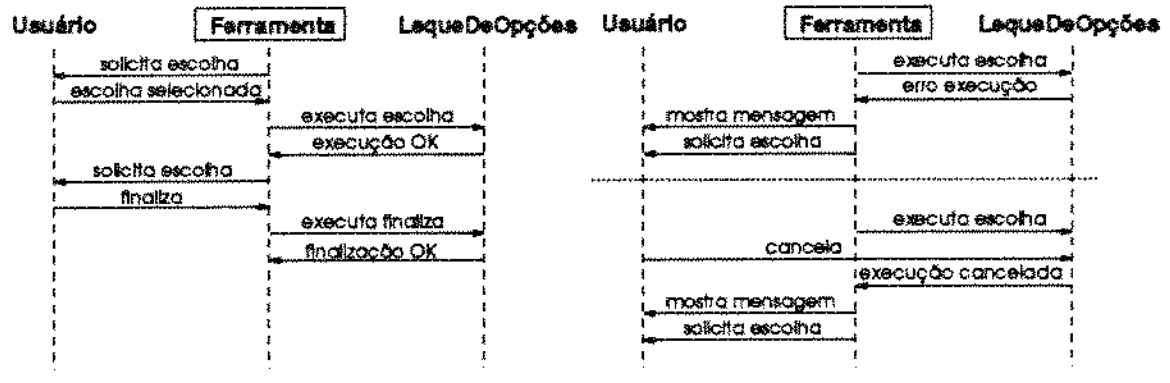


Figura 4.7: *Event-traces* para a classe **Ferramenta**.

Construção dos Diagramas de Estado

Os cenários e os *event-traces* descritos no passo anterior auxiliaram a encontrar as possíveis seqüências de interação da **Ferramenta** com o usuário e da **Ferramenta** com **LequeDeOpções**. O resultado obtido é o diagrama de objetos ilustrado na Figura 4.8.

Dentre as atividades representadas na Figura 4.8, a atividade `executa opção` merece uma atenção especial. O diagrama de estados mais detalhado para esta atividade é ilustrado na Figura 4.9. Trata-se de uma operação que pertence à classe **LequeDeOpções**. A função desta operação é identificar a instância da classe **Opção** que contém a função correspondente à escolha do usuário e invocar sua execução. As

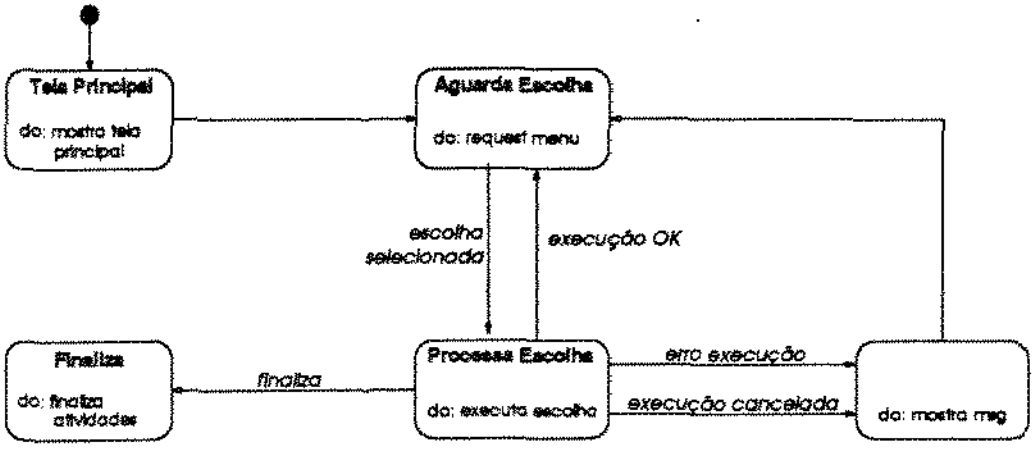


Figura 4.8: Diagrama de estados para classe **Ferramenta**.

opções `op1, op2, ..., opN`, ilustradas na figura, pertencem a um conjunto fictício de instâncias de Opção e as atividades `f1, f2, ..., fN` são as funções correspondentes à escolha do usuário.

Após construir os diagramas de estados é preciso verificar se não existem inconsistências, principalmente em relação aos eventos compartilhados entre eles. A ajuda de uma ferramenta na verificação automática destes detalhes reduz muito o esforço de detecção das inconsistências. Infelizmente, durante o processo de modelagem da plataforma não estava disponível uma ferramenta desse tipo.

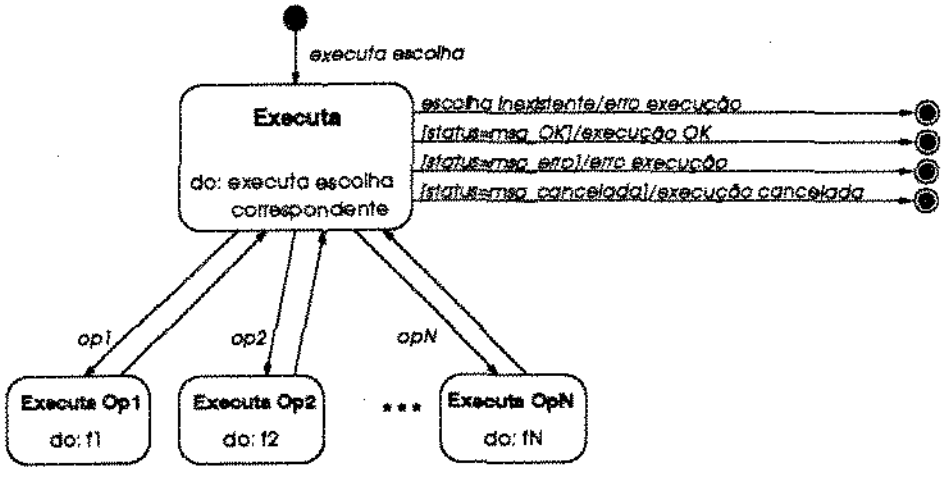


Figura 4.9: Diagrama de estados da operação `executa escolha`, pertencente à classe **LequeDeOpções**.

4.1.5 Modelo Funcional

O Modelo Funcional (MF) descreve como as saídas são processadas a partir de entradas. A seqüência destes processamentos é definida no Modelo Dinâmico. No MF, as dependências funcionais do sistema são descritas através de Diagramas de Fluxo de Dados (DFDs). Quando é preciso entrar em mais detalhes sobre as funções, pode-se usar outras representações, tais como descrições textuais, equações matemáticas e pseudocódigo.

Duas das entidades presentes em um DFD são processos e fluxos. Inicialmente, estas entidades podem ser obtidas pela observação dos outros modelos já construídos. Por exemplo, atividades e ações em diagramas de estados sugerem alguns dos possíveis processos a serem representados em DFDs; os dados passados como parâmetros de um evento equivalem aos dados transportados nos fluxos de entrada e saída dos processos (por exemplo, o evento `escolha selecionada`, na Figura 4.8, entre **Ferramenta** e **LequeDeOpções** tem como parâmetro a identificação da opção escolhida). Portanto, existe uma forte relação entre os elementos presentes nos modelos.

Identificar Dados de Entrada e Saída dos Processos

A Figura 4.10 ilustra uma pequena amostra das trocas de dados entre o sistema e o usuário. Basicamente, as informações que o usuário fornece ou recebe referem-se aos atributos dos componentes ou parâmetros para transformações executadas sobre o diagrama em questão. Estas transformações dependem essencialmente do tipo de FBD. Por exemplo, para um editor, as principais interações são as entradas de dados para os atributos dos componentes do diagrama, enquanto que para um simulador, pode ser os estímulos para animação do diagrama.

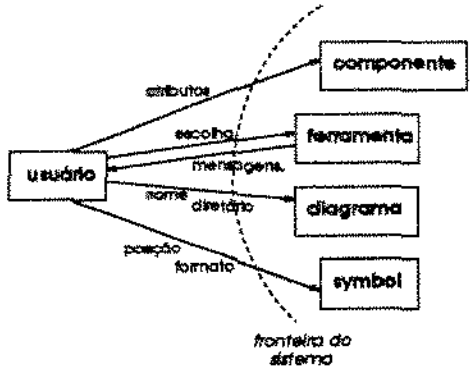


Figura 4.10: Troca de dados entre usuário e aplicação.

Construir os Diagramas de Fluxo de Dados

A construção dos DFDs deve seguir uma estratégia *top-down*, ou seja, inicialmente definem-se os processos mais genéricos e, a medida que houver um melhor entendimento de sua função, eles são detalhados. Os refinamentos prosseguem até que todos os processos tenham um significado relativamente trivial e sem ambigüidades de interpretação. Para as funções que permanecerem obscuras, pode-se complementar os diagramas com informações sobre: os dados de entrada e saída da função, transformações realizadas sobre as entradas para geração das saídas bem como restrições que afetam o processamento da função.

As funções a seguir representam um dos possíveis conjuntos de opções de manipulação do diagrama. As funções que estão em *itálico* afetam apenas as informações gráficas do diagrama. Estas funções foram escolhidas, pois exercitam várias operações definidas nas classes da plataforma. Mais adiante elas estão descritas na forma de DFDs. As funções consideradas são:

- busca diagrama
- salva diagrama
- cria componente
- remove componente
- modifica componente
- consulta componente
- *move componente*
- *modifica símbolo* (modifica valores dos atributos)
- *move símbolo*
- *modifica símbolo graficamente*
- finaliza

Como um componente pode ser constituído de vários símbolos, as funções `modifica símbolo`, `move símbolo` e `modifica símbolo` graficamente devem permitir alterá-los isoladamente.

Algumas observações importantes acerca dos DFDs apresentados nas Figuras 4.11 e 4.12 são:

- fluxos sem rótulo e com origem ou destino em depósito de dados indicam que a informação que flui é todo o conteúdo do depósito;
- fluxos tracejados são chamados de fluxos de controle (o seu uso é desencorajado no método OMT, pois aspectos relacionados com controle são melhor capturados no MD);
- o depósito de dados `Representação Externa` equivale ao depósito de dados `Diagrama`, porém armazenado em algum repositório, por exemplo, em arquivo.

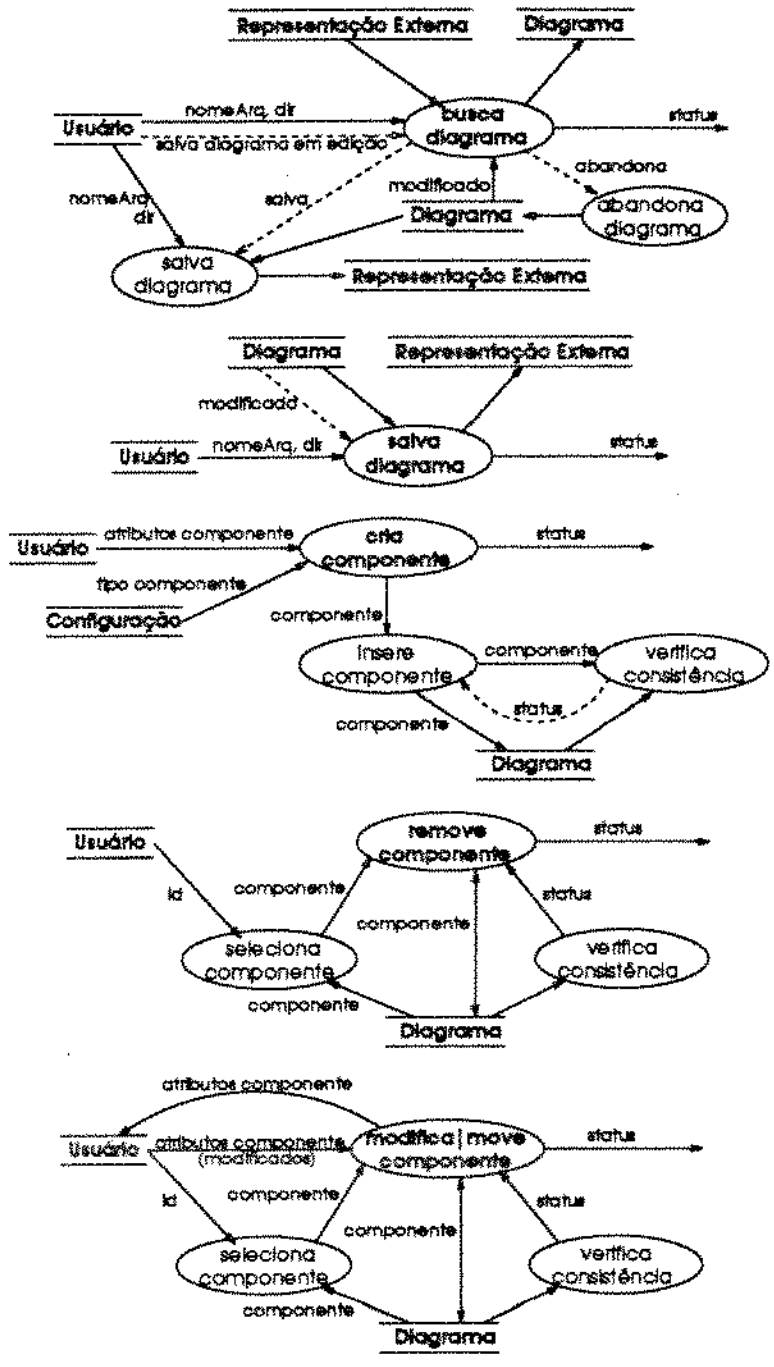


Figura 4.11: DFD para as funções busca diagrama, salva diagrama, cria componente, remove componente, e [modifica componente & move componente].

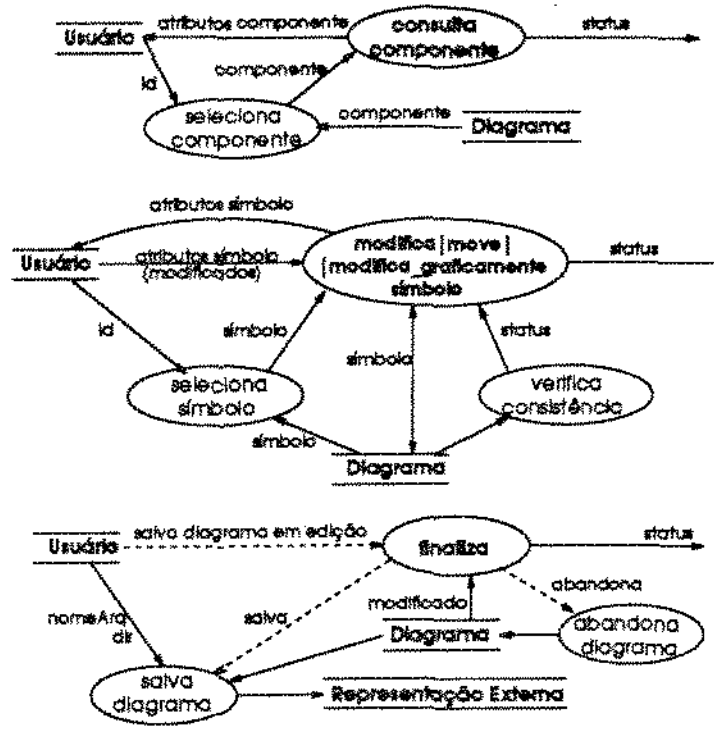


Figura 4.12: DFD para as funções `consulta componente`. [`modifica símbolo` & `move símbolo` & `modifica símbolo graficamente`] e `finaliza`.

4.1.6 Incluir Operações

Continuando o processo de modelagem, a inclusão de operações no diagrama de objetos depende muito dos resultados produzidos pelo MD e MF. Alguns exemplos comuns de operações envolvem a leitura, consulta, alteração e processamento dos valores assumidos pelos atributos da classe. É difícil afirmar se o conjunto de operações e atributos de uma classe foi totalmente definido ou não. Por isso, o MO está sempre sujeito a inclusão de novas características ao longo do processo. Certas operações e atributos são compartilhados por todos os objetos de uma classe como, por exemplo, operações para criação e destruição de um objeto. No diagrama de objetos estas operações e atributos aparecem marcados com um "$" antes de seu nome.

O diagrama de objetos resultante da fase de Análise é ilustrado na Figura 4.13. Nele já estão incluídas as operações. Foi acrescentado também uma associação entre **Diagrama** e **Componente** denominada *corrente*, que serve para guardar a referência a um componente selecionado através da operação `seleciona` em **Diagrama**. Esta

operação deve percorrer o conjunto de componentes do diagrama e verificar qual deles retorna verdadeiro após a chamada da operação compara, em **Componente**. Exemplos de instâncias para as classes **LequeDeOpções** e **Opção** é apresentado no próximo capítulo (Figura 5.4). As operações `salva` e `busca` da classe **Diagrama** possuem um comportamento equivalente às operações de mesmo nome na classe **Representação Externa**.

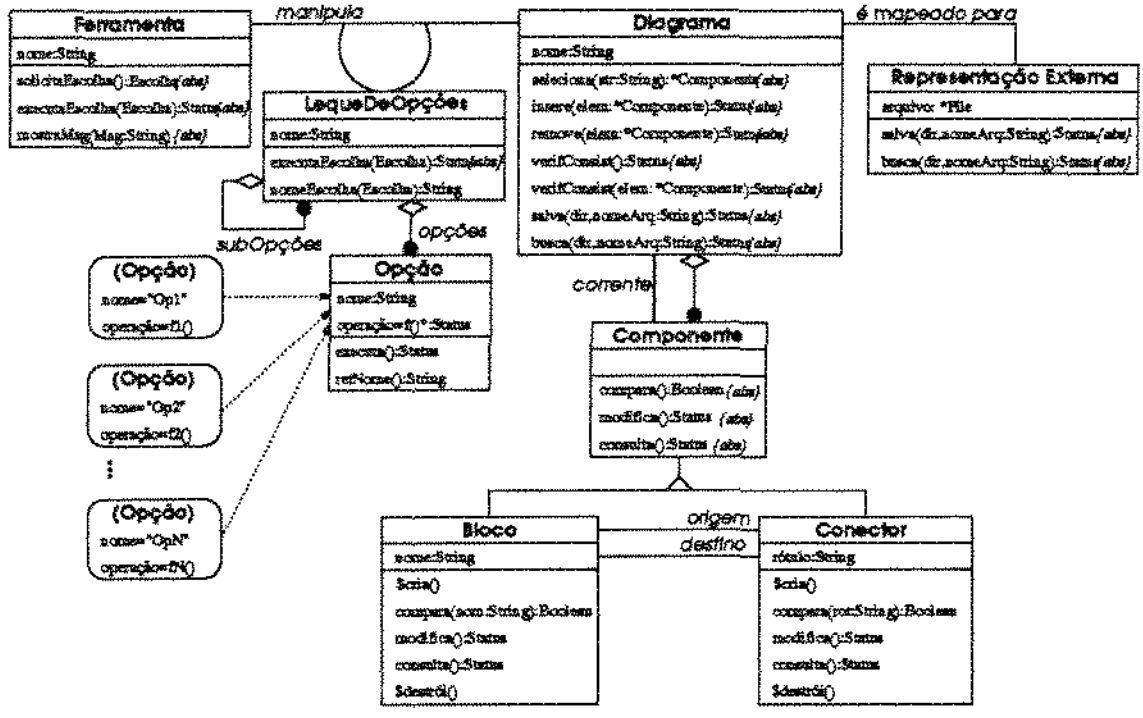


Figura 4.13: Diagrama de objetos contendo as operações identificadas na Análise.

Ainda na Figura 4.13, o tipo *Status*, que aparece como valor de retorno de várias operações, corresponde ao estado final do processamento realizado pela operação. Se algo anormal aconteceu durante o processamento, então a indicação do erro correspondente é retornada como *Status*. Caso contrário, o resultado "OK" sinaliza o sucesso no processamento. Uma alternativa seria centralizar a geração de todas as mensagens de erro em uma única classe e associar a cada mensagem um código. Assim, *Status* passaria a ser um código de erro. Esta alternativa é problemática na medida em que novas indicações de erro são introduzidas, além de necessitar um controle dos códigos usados e comprometer o mecanismo de reutilização. A sintaxe de uma operação no diagrama de objetos é a seguinte:

`nome(parâmetros)[:tipo do resultado]` [{*abs*}] [{*private*}]

Os colchetes indicam que a informação é opcional. Operações sucedidas por {*abs*} são chamadas de *operações abstratas* (na linguagem C++, uma operação desta natureza é chamada de método virtual). São operações cuja implementação só é definida nas classes especializadas. Classes que possuem operações abstratas não podem ser instanciadas e são chamadas de *classes abstratas*. As operações cujo acesso é apenas interno à classe são chamadas de operações privadas. Para distingui-las das operações públicas usa-se o qualificador {*private*}.

4.2 Projeto do Sistema

No método OMT, a fase de Projeto é realizada em duas etapas: Projeto do Sistema e Projeto dos Objetos. Esta seção aborda a primeira etapa, responsável pelas decisões de caráter estratégico, tais como a estrutura do sistema, os recursos necessários e as prioridades.

Durante o Projeto do Sistema a arquitetura do sistema é dividida em vários subsistemas. Isto permite que diferentes membros ou grupos da equipe desenvolvam diferentes subsistemas simultaneamente, além de proporcionar uma abordagem sobre partes menores do problema. Esta atividade é indispensável em aplicações relativamente complexas. Uma boa divisão resulta em subsistemas com alta coesão e baixo acoplamento. Conseqüentemente, a necessidade de interação entre indivíduos de diferentes equipes é bastante reduzida, situação desejável durante o desenvolvimento de um sistema. A seguir, são descritos os passos desta etapa.

4.2.1 Identificar os Subsistemas

Embora o conjunto de classes da plataforma seja relativamente pequeno, a tendência é o aumento da quantidade de classes na medida em que forem acrescentadas as especializações. Isto poderá comprometer a visualização e o entendimento de todo o conjunto de classes, se não houver uma subdivisão prévia.

O processo de divisão pode ser vertical (em camadas) ou horizontal (partições). Em geral, o que acontece é uma combinação destas duas formas, conforme ilustra o diagrama de blocos da Figura 4.14. A figura representa uma aplicação do tipo FBD, onde estão presentes os componentes relevantes ao seu contexto, inclusive os recursos básicos como Hardware, Sistema Operacional e Bibliotecas de Suporte (*toolkits* para interface, bibliotecas gráficas para desenho do diagrama). Neste tipo de representação, quando dois blocos estão adjacentes verticalmente, então existe uma interação entre eles. A camada inferior presta serviços para a camada superior. A

interação entre blocos adjacentes horizontalmente é geralmente fraca ou inexistente.

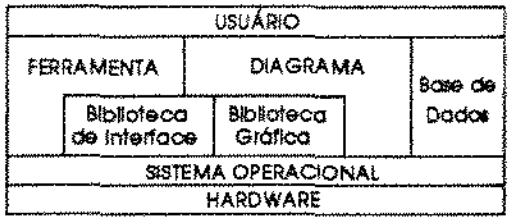


Figura 4.14: Diagrama de blocos para uma ferramenta baseada em diagrama.

No topo, interagindo com a Ferramenta, o Diagrama e a Base de Dados, está o usuário do sistema. A Base de Dados é apenas o repositório de informações de diagramas específicos. Também fazem parte do bloco Diagrama os seus componentes e símbolos gráficos. O Usuário interage com a Ferramenta escolhendo as opções de manipulação do Diagrama. Entre Usuário e Diagrama são trocadas informações sobre atributos relativos ao diagrama manipulado como um todo ou de seus componentes (incluindo os símbolos gráficos). Com relação ao Usuário é permitido que ele leia e altere as informações armazenadas em Base de Dados. Para isto é necessário que o formato de armazenamento seja legível. Mais detalhes sobre este tema são apresentados na próxima seção.

4.2.2 Gerenciamento dos Dados Armazenados

O armazenamento de dados serve, entre outras coisas, para dar um certo grau de persistência às informações. Dentre as alternativas de gerenciamento destes dados, a escolha depende basicamente da complexidade dos acessos, volume de dados envolvido e exigências com relação à presença de mecanismos para segurança no acesso das informações e controle de consistência. Quanto mais recursos forem necessários, maiores serão os custos envolvidos. Soluções muito sofisticadas como os Sistemas Gerenciadores de Banco de Dados (SGBDs) aumentam a complexidade do sistema desnecessariamente se arquivos forem suficientes. No caso de sistemas orientados a objetos as soluções mais adequadas têm sido os Bancos de Dados Orientados a Objetos (por exemplo, Postgres, VODAK, Prologic XL). Eles possuem conceitos muito importantes como *objetos persistentes*, os quais facilitam consideravelmente a tarefa de armazenamento de objetos.

No caso da plataforma, os acessos aos dados armazenados restringem-se às operações de escrita e leitura de todo o diagrama. O volume de dados armazenados não é tão grande. O conteúdo das informações não requer sigilo ou proteções contra acessos indevidos. Mecanismos de controle para acessos simultâneos às informações não são

essenciais. No caso das FBDs, o armazenamento em arquivos é suficiente. O uso de SGBDs e Banco de Dados Orientado a Objetos pode até ser viável, desde que o *overhead* de gerenciamento não afete significativamente a execução do sistema ou aumente a complexidade da implementação.

A adoção de um arquivo texto para armazenamento das informações do diagrama requer uma linguagem que defina a sintaxe do diagrama. É desejável também que as informações descritas na linguagem sejam de fácil entendimento mesmo sem o uso de ferramentas, ou seja, que uma pessoa possa ler um arquivo e identificar, entender e eventualmente modificar as informações nele contidas. A seguir é apresentado um exemplo de linguagem definida conforme os requisitos anteriores. A sintaxe é uma adaptação da linguagem definida por Paulisch [PT90], chamada GRL (Linguagem para Representação de Grafos). Baseado nesta sintaxe, ferramentas como **lex** e **yacc** podem ser usadas para facilitar a sua implementação. A sintaxe considera o diagrama como um conjunto de atributos e componentes. Cada componente, por sua vez, é um conjunto de atributos e componentes (permite estabelecer elos de ligação entre componentes, por exemplo, uma hierarquia). Os atributos são informações simples, do tipo numérico, alfanumérico, seqüências de atributos ou objetos da classe **Symbol**, que podem ter uma coleção de atributos, inclusive do tipo `simbolo`.

```
diagrama          ::= \begin{diagrama}
                        [atributo]*
                        [componente]*
                      \end{diagrama}
atributo          ::= nome_atributo = valor
nome_atributo     ::= letra [letra|digito|_]*
letra             ::= a..z,A..Z
digito            ::= 0..9
valor             ::= numero
                    | caracter
                    | cadeia_caracteres
                    | vetor
                    | simbolo
componente        ::= \begin{componente}
                        [atributo]*
                        [nome_atributo = componente]*
                      \end{componente}
numero            ::= qualquer numero inteiro ou real
caracter          ::= qualquer caracter ASCII
cadeia_caracteres ::= " [caracter]* "
vetor             ::= { [valor]* }
simbolo           ::= \begin{simbolo}
                        [atributo]*
                      \end{simbolo}
```

Para fazer o mapeamento entre os dados de um arquivo e os atributos definidos nas especializações das classes **Diagrama**, **Componente** e **Symbol**, existe uma classe com esta finalidade chamada **Representação Externa**. Esta classe é responsável pela leitura (operação `busca`) e gravação (operação `salva`) dos dados.

Algumas considerações desejáveis ao armazenamento de um diagrama podem ser úteis. A ordem de armazenamento e leitura dos atributos das classes não deve ser rígida. Para evitar o armazenamento desnecessário de informações, é desejável que os atributos com valores *default* sejam omitidos [PT90]. Se o mesmo atributo de um objeto estiver definido mais de uma vez, assume-se apenas o valor da última atribuição.

4.2.3 Tratamento de Situações Não Usuais

São situações pouco freqüentes ou imprevisíveis, que ocorrem durante a execução do sistema. Exemplos típicos são: inicialização, finalização e falha do sistema. As duas primeiras são previsíveis, porém a terceira é indesejável. Como é difícil garantir que um sistema estará livre de erros, é preciso planejar previamente quais providências deverão ser tomadas no sentido de contornar de forma adequada as situações de erro. Caso uma situação de erro acarrete o término da execução do sistema, é importante que as informações relevantes sejam armazenadas, inclusive aquelas que possam ajudar na detecção do defeito.

Para as aplicações do tipo FBD, a inicialização da classe **Ferramenta** (no caso a operação `$inicializa`) deve preparar o ambiente de interação com o usuário, deixando disponíveis as opções de manipulação de um diagrama e uma área para visualização e/ou manipulação do seu desenho. A finalização da classe Ferramenta (operação `$finaliza`) deve verificar se o diagrama manipulado foi modificado e está sendo abandonado sem atualização das informações em arquivo. Se isto ocorrer, o sistema deve notificar o usuário e permitir a atualização. De um modo geral, as "situações casuais" para as FBDs são relativamente simples. No caso de um erro ter ocorrido, a causa deve ser propagada pelas operações até chegar à ferramenta, que se encarregará de transmitir ao usuário a mensagem de erro correspondente (operação `mostraMsg`). Na plataforma, o retorno de um processamento é denominado *Status*.

4.2.4 Arquitetura da Plataforma

As arquiteturas de diversos tipos de sistemas, como os de processamento em *batch*, interativos, tempo real, simuladores, e gerenciadores de transações, diferem significativamente umas das outras. Dependendo das características do sistema, um modelo

(MO, MD ou MF) pode ser mais relevante. Por exemplo, o Modelo Dinâmico para um sistema interativo é essencial, enquanto que para um sistema de processamento em *batch* pode ser desnecessário.

Na plataforma, sua arquitetura combina subsistemas que possuem diferentes características e, portanto, não há uma uniformidade na importância dos modelos. A representação gráfica da arquitetura não é definida precisamente no livro de Rumbaugh **et al.** Por isso podem existir pontos obscuros no diagrama construído. As regiões tracejadas não fazem parte da definição da arquitetura, mas dão uma visão dos módulos enfocados em cada modelo descrito neste capítulo.

Conforme a Figura 4.15, a arquitetura da plataforma é composta por dois subsistemas: Ferramenta e Diagrama. Os módulos Representação Externa, Componente e Symbol são integrantes de Diagrama. Ferramenta é um híbrido de sistema interativo com gerenciador de transações. Seu papel é interagir com o usuário e capturar a opção a ser processada. As linhas pontilhadas ligando Opção a outros módulos indicam os responsáveis pelo processamento da opção, que pode ser Representação Externa, Componente, Symbol ou Diagrama.

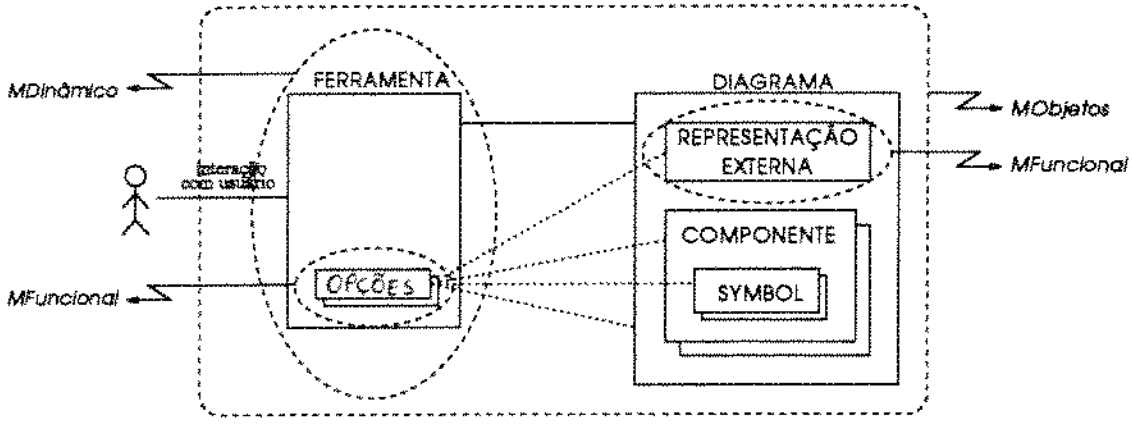


Figura 4.15: Arquitetura da plataforma.

Em relação às linhas tracejadas da Figura 4.15, todos os módulos, ou classes, são capturados pelo MO. Por ter um comportamento mais uniforme, o MD é especificado apenas para a **Ferramenta** (Seção 4.1.4), sua funcionalidade é relativamente simples. O comportamento e a funcionalidade das classes **Diagrama**, **Componente** e **Symbol** são mais abstratos e, portanto, são mais precisamente definidos nas especializações. Por último, o MF captura algumas das funcionalidades possíveis para as opções do **LequeDeOpções** (Seção 4.1.5) e o mapeamento do diagrama para uma base de dados, realizado por Representação Externa (Seção 4.2.2 e 4.3.1).

4.3 Projeto de Objetos

A identificação dos atributos e operações não é um atividade trivial. É um processo de descoberta que evolui na medida em que os conceitos da aplicação passam a ser melhor entendidos. Por este motivo, esta fase de desenvolvimento visa a refinar os modelos construídos na Análise, incluindo detalhes que anteriormente não eram relevantes, mas que são fundamentais para a fase de Implementação. Os refinamentos devem obedecer aos compromissos e decisões da fase anterior (Projeto do Sistema). Como as restrições estabelecidas anteriormente não são tão rígidas, isto facilita as tarefas da fase aqui descrita.

As atividades desta etapa envolvem a criação de novas classes, atributos, operações e associações mais relacionadas com aspectos de implementação; definição de estruturas de dados e algoritmos; bem como a escolha dos detalhes sobre a implementação de associações. Outras atividades mencionadas em [RBP+91] não serão mencionadas, pois são relevantes apenas em outros contextos.

Com a criação de novas classes, atributos, operações e associações, continuam sendo válidas as recomendações da Análise sobre a redução de redundâncias e compartilhamento de características em comuns, através de herança.

4.3.1 Projeto dos Algoritmos

Para as operações mais complexas, deve-se descrever sua funcionalidade através de algoritmos. Assim, a tarefa de implementação torna-se mais fácil e menos sujeita a ambigüidades. Ao definir os algoritmos, a conseqüência é identificar operações relevantes a um tipo de aplicação ou uma forma de implementação em particular. Para evitar a definição excessiva de operações desse tipo, este passo da fase de Projetos não será detalhado. Apenas o exemplo do mapeamento que a classe **Representação Externa** faz de um diagrama para um arquivo é dado no algoritmo a seguir:

```
classe RepresentaçãoExterna {
   Diagrama *diagrama
   File *arq
   String nome_arq, dir_arq
   operações públicas:
      busca (nome,dir:String):  Status
      salva (nome,dir:String):  Status
   operações privadas:
      abre_arq (modo:String) /* modo:  leitura, escrita, ambos */
```

```
      fecha_arq ()
}

Status RepresentaçãoExterna::busca(nome,dir) {
   arq ← abre_arq ("leitura")
   busca atributos diagrama do arquivo
   associa valores atributos ao diagrama
   enquanto (FIM_ARQUIVO = Falso) faça
      caso (tipo componente) seja
         TipoX:
            busca atributos componente arquivo
            busca informações estruturais arquivo
            cria componente
            associa valores atributos componente
            insere componente no diagrama
         TipoY:
         ...
         default:
            retorna ("componente não identificado")
      fim_caso
   fim_enquanto
   fecha_arq ()
   retorna ("OK")
}

Status RepresentaçãoExterna::salva(nome,dir) {
   arq ← abre_arq ("escrita")
   lê atributos diagrama
   salva atributos diagrama
   componente ← diagrama.retPrimeiro ()
   enquanto (componente != Null) faça
      caso (tipo componente) seja
         TipoX:
            lê atributos componente
            salva atributos componente
            lê informações estruturais
            salva informações estruturais
         TipoY:
         ...
```

```
            default:
               retorna ("componente desconhecido")
         fim_caso
         componente ← diagrama.retPróximo ()
      fim_enquanto
      fecha_arq ()
      retorna ("OK")
}
```

Neste passo da fase de Projeto, é preciso definir vários algoritmos. Porém, em alguns casos, é necessário usar algoritmos "clássicos" como *quicksort*, busca em árvores, desenhos de curvas, métodos de cálculo numérico e assim por diante. Alguns deles são escolhidos por questões de eficiência, outros pela complexidade dos problemas que resolvem. No caso das FBDs, pelo menos dois algoritmos podem ser úteis: desenho de curvas, para interligar os blocos e desenho automático de diagramas. O primeiro deles encontra-se implementado nas ferramentas desenvolvidas no ICMSC/USP [Can93, FM92, TM91, Bat91], baseado no método utilizado chamado b-spline [NS87, Har87b]. O segundo, por questões de complexidade do tema, não será aprofundado. Existe uma tese de mestrado no DCC/Unicamp [Sil94] que trata especificamente deste assunto. Mais referências bibliográficas são apresentadas e comentadas na Seção 6.4.

No sentido de promover a extensibilidade das ferramentas construídas com o uso da plataforma, é importante ressaltar que a legibilidade dos algoritmos é um fator relevante, mesmo que haja perdas em eficiência. A propósito, em benefício de características como extensibilidade e facilidade de reutilização, os sistemas orientados a objetos podem ser até um pouco menos eficientes quando este fator não for crítico.

Seleção das Estruturas de Dados

Durante a Análise, o enfoque era dado somente à estrutura abstrata das informações, mas, na fase de Projeto, é preciso decidir como elas serão organizadas. Exemplos de estruturas para a organização de dados são: vetores, listas, filas, pilhas, conjuntos, tabelas de *hashing*, árvores e variações destas estruturas. Em algumas linguagens de programação, a utilização dessas estruturas é facilitada, pois já fazem parte do conjunto de bibliotecas predefinidas.

Uma das situações, onde as estruturas de dados são úteis, é na implementação das associações com multiplicidade "um-para-vários" e "vários-para-vários." Mais detalhes são apresentados na Seção 4.3.3. Para a plataforma, é preciso uma estrutura

de dados para organizar os componentes de um diagrama. Componentes, por sua vez, são especializados em blocos e conectores. A coleção de blocos e conectores pode ser disposta em uma Lista. A princípio, para cada tipo de componente do diagrama deve-se identificar qual a estrutura mais adequada para ela. A seguir é apresentada, a título de exemplificação, a definição da classe List. Antes porém, são definidas as convenções e a estrutura dos tipos usados pela estrutura. O símbolo $< T >$ é um Template que representa o tipo do elemento a ser armazenado na estrutura.

```
struct Element {el:*< T >; next:*Element; previous:*Element;};

struct KeyType {code:Integer; identifier:String; position:Coord;
region:Coord[2]};

enum Orientation = {next, previous};
```

*** Estrutura de Dados: List ***

classe
List< T >
atributos
first:*Element
current:*Element
operações

$create()		/*cria lista vazia*/
empty():Boolean		/*verifica se lista está vazia*/
insert(e:*< T >, Orientation):Boolean	where:	/*insere "e" na posição indicada por "where", que pode ser [first/last/next/previous], "current" passa a ser "e", retorna True se sucesso*/
delete(e:*< T >):Boolean		/*remove elto corrente, retorna True se sucesso*/
update(e:*< T >):Boolean		/*troca o elto corrente por "e", retorna True se sucesso*/
retrieve():*< T >		/*retorna o elto corrente, ou Null se $\nexists$*/
goto(where:Orientation): Boolean		/*posiciona elto corrente conforme indicado por "where", neste caso pode ser [first/last/next/previous], retorna True se $\exists$*/
find(key:KeyType):Boolean		/*posiciona elto corrente cuja identificação coincide com "key", retorna True se achar*/

Definição de Classes e Operações Internas

Ao atingir esta etapa da fase de Projeto, as classes, atributos e operações da aplicação já foram identificadas. Neste ponto, algumas classes e operações não mencionadas explicitamente na descrição do problema precisam ser criadas para auxiliar na implementação das operações já definidas. Neste contexto, estão as estruturas de dados citadas na seção anterior, a classe **Configuration** e a classe **Symbol**. As estruturas de dados auxiliam na implementação das associações com multiplicidade maior que um. As classes **Configuration** e **Symbol** servem para capturar aspectos gráficos de diagramas.

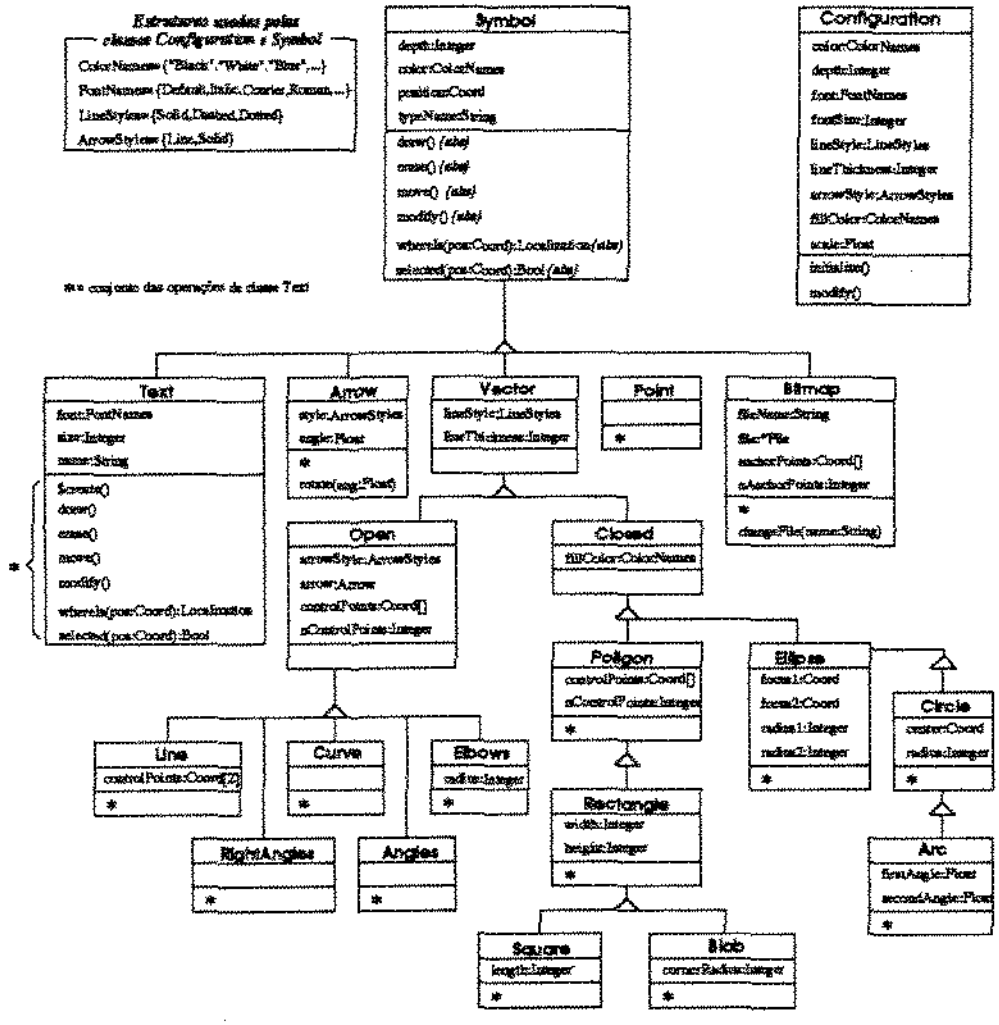


Figura 4.16: Hierarquia de objetos gráficos e a classe **Configuration**

A Figura 4.16 ilustra a hierarquia de símbolos gráficos e a classe **Configuration**.

Para evitar a repetição dos nomes das operações nas várias classes da hierarquia, onde se vê um asterisco (*) significa que as operações abstratas da classe **Symbol** são redefinidas e estão associadas a uma implementação específica, conforme ilustrado na classe **Text**. Algumas classes podem possuir operações adicionais, tais como **Arrow** e **Bitmap**. As classes **Symbol**, **Vector**, **Open** e **Closed** são classes abstratas.

O comentário que segue vale para todas as classes da plataforma e diz respeito ao conceito de operações públicas e privadas. Visando à extensibilidade das classes, é muito importante que a definição das operações públicas e de suas interfaces (lista de parâmetros de entrada e saída) permaneçam estáveis, ou seja, que não se alterem. O custo de uma mudança na interface de uma operação pode ser muito alto, pois interfere em todas as classes associadas a ela. Assim, a definição de operações públicas deve ser feita com muita cautela. No caso das operações privadas, as conseqüências não são tão graves, pois o escopo das alterações é apenas o da própria classe em questão. Por exemplo, se a hierarquia de símbolos gráficos e suas operações forem bem definidas, as conseqüências de uma mudança na implementação das operações de desenho de uma biblioteca gráfica para outra seriam mínimas. Do ponto de vista das classes que requisitam seus serviços, nada precisaria ser alterado.

Na Figura 4.17 é apresentado o Modelo de Objetos completo da plataforma, contendo as novas operações e atributos, inclusive os atributos e as estruturas de dados que dizem respeito à implementação dos relacionamentos (associação e agregação). Neste estágio da modelagem, muitos detalhes, inclusive de implementação, são apresentados. O desenho das caixas com bordas arredondadas, apontando para a classe **Opção**, é um recurso adicional da notação para representar algumas das possíveis instâncias de objetos para a classe indicada.

4.3.2 Ajuste das Classes para a Adoção do Mecanismo de Herança

As recomendações deste passo foram muito usadas durante a modelagem da plataforma ([RBP+91], página 242). Elas são muito importantes, pois alertam sobre equívocos cometidos no uso incorreto do mecanismo de herança.

Muitas vezes a herança é empregada como uma técnica de implementação, onde o objetivo maior é aumentar o compartilhamento. Mas isto não é recomendado, pois algumas características herdadas podem não ter um significado na classe especializada. Operações herdadas desnecessariamente podem até causar efeitos indesejáveis quando solicitadas. Nesse caso, a solução é o uso de delegação. A técnica de delegação efetivada por agregações permite que apenas as operações relevantes sejam compartilhadas. Um exemplo desse tipo de situação é o das estruturas de dados

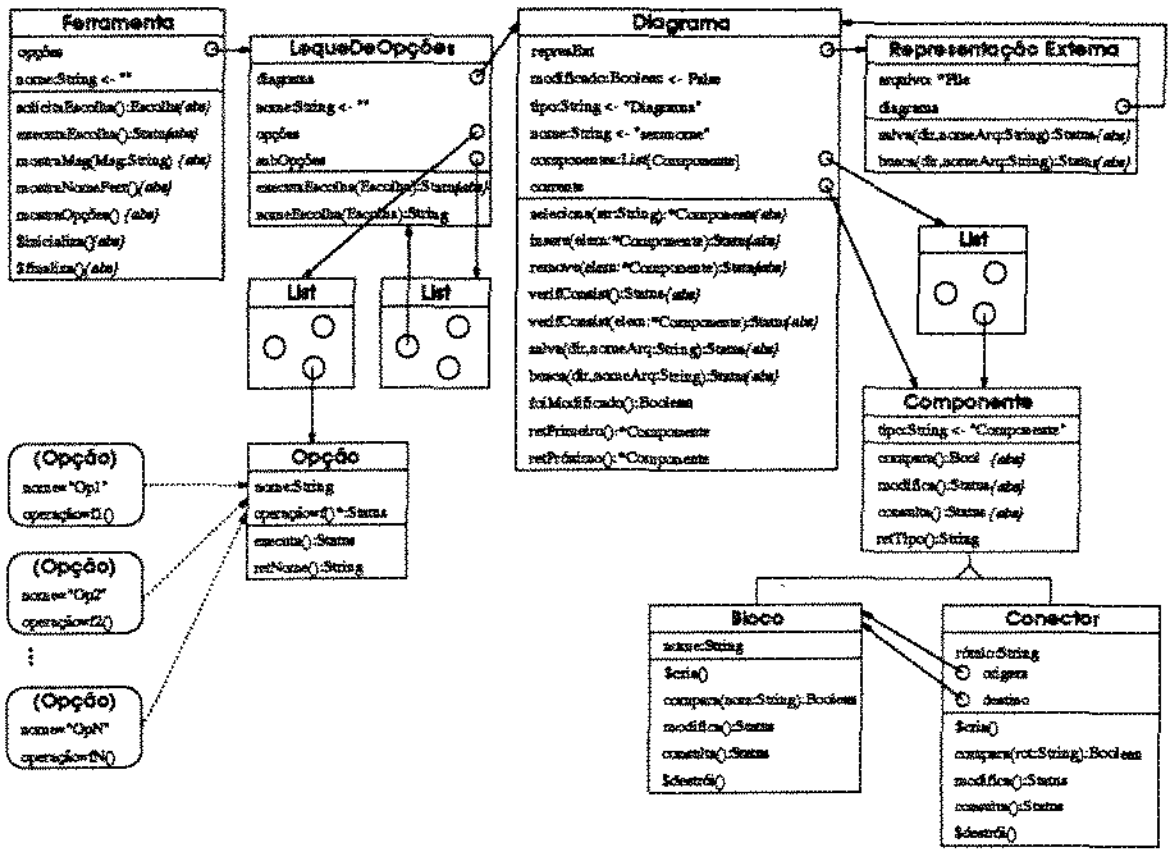


Figura 4.17: Modelo de Objetos completo da plataforma.

Lista e Pilha. O procedimento natural é fazer com que a classe Pilha herde as características de Lista. Entretanto, ao se fazer isto não só as operações `empilha` e `desempilha` passam a estar disponíveis como também as operações `insere` e `remove`, as quais são indesejáveis para a classe Pilha, pois permitem a inserção e remoção de elementos em qualquer posição da estrutura. A Figura 4.18 ilustra a representação destas duas situações. No caso recomendado, as operações `empilha` e `desempilha` fazem uso de `insere` e `remove`, mas o acesso a estas duas últimas é vedado para um usuário externo de Pilha. Uma forma de checar se é recomendável que duas classes se relacionem por herança é verificando se a classe descendente é de fato um subtipo da classe ancestral.

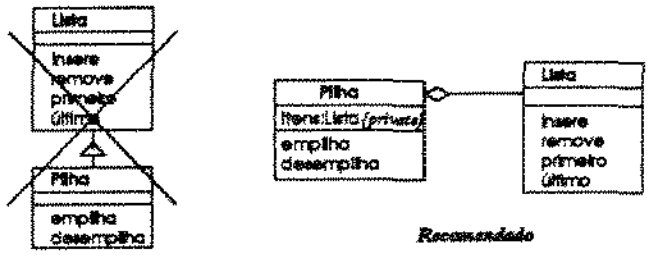


Figura 4.18: Duas formas de modelagem da estrutura **Pilha**, a partir de **Lista**

4.3.3 Projeto das Associações

As associações indicam os caminhos de acesso na comunicação entre objetos. Pelo ponto de vista de implementação, é importante saber qual a direção do relacionamento (unidirecional ou bidirecional), em termos de consultas, bem como a sua multiplicidade.

Quando a multiplicidade for 1, a associação pode ser implementada como um ponteiro – atributo contendo uma referência a outro objeto. A Figura 4.17 já mostra estas associações como ponteiros. Se a multiplicidade for maior que 1, então o atributo possivelmente faz referência a um conjunto de ponteiros, que podem ser estruturas do tipo lista, conjunto, tabelas de hashing, ou até vetores simples. Uma outra alternativa seria uma implementação na forma de objeto.

Para a plataforma foi estabelecido que uma Lista seria adequada à implementação do relacionamento entre diagrama e componente, pois não existem restrições que requeiram estruturas mais sofisticadas. No exemplo do próximo capítulo, é definida a estrutura Árvore, que é a mais adequada para organização de um dos componentes do tipo de diagrama escolhido para ser especializado a partir da classe **Diagrama**.

Para associações que possuem atributos, como é o caso da relação entre **Ferramenta** e **Diagrama**, a solução é implementá-la como uma classe independente, denominada **LequeDeOpções**. Na classe **Ferramenta** há um ponteiro para **LequeDeOpções** e na classe **LequeDeOpções** há um ponteiro para **Diagrama**. A classe **LequeDeOpções** é considerada um atributo da associação pois ela é uma propriedade dessa associação e não das classes relacionadas pela associação.

4.4 Comentários

A referência básica para este capítulo é o livro que descreve o método OMT [RBP+91], usado no desenvolvimento da plataforma. Uma característica do método é o esforço

dedicado às fases de Análise e Projeto. Isto reduz significativamente o número de decisões a serem tomadas na fase de Implementação e dessa forma, possibilita que muitas das incorreções sejam eliminadas ainda no processo de modelagem.

Capítulo 5

Um Exemplo de Utilização da Plataforma

Este capítulo define um procedimento de modelagem de ferramentas manipuladoras de diagramas, com auxílio da plataforma, seguido de um exemplo de seu uso. Este exemplo serve para mostrar qual é a participação da plataforma durante o processo de modelagem e verificar as vantagens e as desvantagens proporcionadas.

A aplicação escolhida foi um Editor Gráfico de Estadogramas. O emprego da plataforma no desenvolvimento dessa ferramenta é no provimento de classes básicas para definição de dois elementos principais: editor gráfico e estadograma, isto é, a ferramenta e a classe alvo de objetos por ela manipuláveis. O processo de definição consiste basicamente em atividades de identificação das particularidades dos elementos a serem construídos e na especialização das classes predefinidas.

A decisão pelo editor gráfico permitiu exemplificar como ocorre a separação das informações gráficas e estruturais do diagrama. Ao isolar estas informações é possível ter um reaproveitamento maior dos dados manipulados e armazenados, inclusive por ferramentas não gráficas.

A decisão por estadogramas se deu, principalmente, por três razões. A primeira refere-se à maior expressividade das construções de estadogramas, comparado com a de diagramas de transição de estados na representação do comportamento de sistemas reativos, das interações entre objetos de sistemas OO, na especificação de interfaces, além de outras aplicações. Portanto, trata-se de uma representação gráfica de grande importância no desenvolvimento de sistemas. Outra razão foi devido à necessidade de se exercitar a extensibilidade de classes derivadas da plataforma. Estadogramas possuem diversas variantes na definição de sua estrutura, semântica e apresentação (estados parametrizáveis, transições automáticas, sobreposição de estados). A mo-

delagem de uma ou mais dessas variantes poderia dar uma noção deste requisito da plataforma. A última razão deve-se ao interesse que o Projeto Xchart [Lie93] tem por esta notação. O projeto irá usufruir dos resultados obtidos neste trabalho no processo de criação de ferramentas baseadas em Xchart. Esta notação é derivada de Estadogramas e destina-se à representação do comportamento de sistemas reativos potencialmente distribuídos, em especial, o controle de diálogo de Interfaces Homem-Computador.

Uma descrição resumida da notação de Estadogramas pode ser encontrada no Apêndice B. Maiores detalhes encontram-se nas referências [Har87a] e [HPSS87]. Referências complementares como [iLI89] e [HLN+90b] descrevem uma ferramenta que auxilia no desenvolvimento de sistemas reativos complexos, usando Estadogramas como uma das suas principais notações.

5.1 Procedimento de Modelagem

Antes de iniciar a descrição da modelagem do Editor Gráfico de Estadogramas (EdGE) será definido um procedimento para esta tarefa. O procedimento consiste na construção de três modelos (Modelo de Objetos, Dinâmico e Funcional), conforme propostos no método OMT. As subseções seguintes descrevem como conduzir este procedimento através de etapas agrupadas em duas fases: Análise e Projeto.

5.1.1 Análise

A) Identificar os Objetos

1. Definir a natureza (gráfica ou não) e a função da ferramenta
2. Definir as características "macroscópicas" do diagrama, ou seja:
   (a) quais os elementos que o constituem;
   (b) quais desses elementos são considerados **componentes** e quais são apenas seus **atributos**;
   (c) classificar os **componentes** em dois grupos: **blocos** e **conectores**;
   (d) identificar as figuras geométricas (símbolos) utilizadas na representação dos componentes, caso a ferramenta seja gráfica.

B) Incorporar Extensões na Plataforma

Nesta etapa, os objetos identificados em (**A**) são úteis para a definição das novas classes que estenderão a plataforma. As novas classes são inseridas como especializações em uma das seguintes classes abstratas ou de possíveis subclasses já derivadas: **ferramenta**, **diagrama**, **componente** ou **symbol**.

O nível hierárquico apropriado depende das características de cada classe e das especializações já existentes. Como ainda não existem especializações, o exemplo adotado irá partir das classes abstratas mencionadas acima.

Uma observação importante a esta etapa refere-se à criação de classes intermediárias. São classes adicionais que visam a fatorar as características em comum de duas ou mais classes. Por exemplo, um Diagrama de Atividades [HLN+90b] possui fluxos de controle e de dados. Em vez de especializá-las diretamente da classe **conector**, seria interessante criar uma classe intermediária, chamada **fluxo**, contendo as características comuns aos diversos tipos de fluxos. Através da adoção de medidas como esta, pode-se contribuir para aumentar a capacidade de reutilização de código gerado. Vale ressaltar que esta atitude foi uma das preocupações durante a modelagem da plataforma na busca de **reusabilidade** e **extensibilidade**.

C) Identificar as Relações entre Classes

Em geral, as associações entre as novas classes são as mesmas já existentes no nível das classes abstratas da plataforma. Por isso, em muitos casos esta etapa não traz contribuições adicionais às extensões do modelo de objetos da plataforma.

D) Identificar os Atributos

Neste momento, as principais classes e relações da aplicação já foram identificadas. Agora é preciso começar a detalhar o conteúdo de cada classe, iniciando pelos seus atributos. Recomenda-se que sejam incluídos apenas os atributos cuja identificação é mais imediata. Alguns deles já foram destacados em (**A.2.b**). De posse desses atributos, faz-se uma comparação com os atributos das superclasses a fim de identificar as alterações e complementações necessárias.

Começa-se pela identificação dos atributos das superclasses na hierarquia de ferramenta, em seguida pelas superclasses diagrama e por último em componentes. A modelagem dos símbolos, caso a ferramenta seja gráfica, deve ser deixada para a fase de Projeto. Isto se deve em razão deles serem considerados aqui como classes auxiliares e envolverem aspectos relacionados mais com questões de implementação.

Existem atributos na classe **diagrama** (ou de seus componentes) que são necessários em função do tipo de ferramenta que irá manipulá-los, por exemplo, se a ferramenta for um simulador de DTEs, seus estados precisam de um atributo especial que indique o *status* de ativação. Para um editor gráfico esta informação é desnecessária. Assim, também é importante considerar as características da ferramenta na definição da classe da hierarquia de diagrama que está sendo especializada.

E) Identificar as Operações

As mesmas recomendações feitas em (**D**) para atributos deve ser seguida aqui com um enfoque em operações. A única observação adicional é com relação à classe **LequeDeOpções**, que deve agregar um conjunto de opções, definidas em função do tipo de ferramenta e do tipo de diagrama. As opções definidas passam a ser instâncias da classe **Opção**.

F) Modelagem Dinâmica

O Modelo Dinâmico (MD) descreve o padrão de comportamento da aplicação. Através de interações internas ou externas ao sistema, os objetos sofrem transformações no conjunto de valores dos seus atributos, que são traduzidos em termos de estados e transições em um estadograma. Em algumas FBDs estas interações são triviais. Por isso, apenas os casos mais complexos precisam ser modelados. As tarefas comumente realizadas entre o usuário e as classes da plataforma são:

- usuário - ferramenta: escolha de uma opção, apresentação de resultados ou informações;
- usuário - diagrama: entrada de dados para manipulação (criação, alteração) do diagrama;
- usuário - componente: entrada de dados para manipulação (criação, alteração, seleção) dos componentes; e
- usuário - símbolo: entrada de dados para manipulação (desenho, movimentação, alteração, seleção) de símbolos.

As interações que já estiverem representadas no MD da plataforma não precisam ser novamente definidas. Ao terminar esta etapa é preciso revisar o Modelo de Objetos, a fim de incluir as operações levantadas.

G) Modelagem Funcional

Esta etapa possui um papel muito importante para a fase de Implementação: definir a funcionalidade das operações. Ao final deste processo, faz-se novamente uma revisão do Modelo de Objetos para complementação do MO com as novas operações identificadas.

Recomenda-se a seguinte ordem na construção dos DFDs: **Opção**, **Ferramenta**, **Diagrama** e **Componente**.

5.1.2 Projeto

A fase de Projeto tem como tarefas:

- Identificar as operações auxiliares para as classes;
- Especificar o algoritmo das operações mais complexas;
- Definir as estruturas de dados para organização de cada componente do diagrama;
- Associar uma biblioteca gráfica para implementação dos símbolos usados no diagrama.

5.2 Modelagem do Editor Gráfico de Estadogramas (EdGE)

Com base no procedimento da seção anterior será descrito a sua aplicação na modelagem do Editor Gráfico de Estadogramas.

5.2.1 Análise

A) Identificar os Objetos

Os objetos identificados nesta etapa são os seguintes:[1]

[1] A codificação entre <> corresponde à identificação de um passo particular da etapa **A** que foi seguido na fase descrita.

editor gráfico: ferramenta gráfica cuja função é gerenciar a edição de estadogramas. Define a área onde o usuário irá manipular (desenhar) os elementos do diagrama e fornece acesso ao conjunto de opções de edição < $A.1$ >.

estadograma: diagrama a ser manipulado pela ferramenta. É composto por uma coleção de estados e transições organizadas de acordo com regras sintáticas e semânticas específicas < $A.2$ >.

estado: um dos componentes do diagrama. Diferentemente dos DTEs, estes estados podem ser dispostos em uma hierarquia. Os tipos possíveis são XOR, AND e Básico. A diferença entre eles é a restrição imposta aos seus estados descendentes diretos quando o estado estiver ativo. Enquanto o primeiro só permite que um de seus estados descendentes diretos esteja ativo em um determinado instante de tempo, o outro requer que todos estejam ativos. Estados sem descendentes (nós folhas na árvore de estados) são considerados como sendo do tipo Básico. A forma de representação mais encontrada para os estados é chamada de *blob* ou "bolha," pois é traçado na forma de um retângulo com cantos arredondados. O estado AND é diferenciado graficamente do estado XOR pelos segmentos de reta tracejados dividindo as regiões de seus estados descendentes. Um estado Básico é traçado como um estado XOR. Estados descendentes são contidos nas áreas delimitadas pelo seu ancestral.

transição: outro componente do estadograma. Estabelece as possíveis seqüências de ativações de estados. É um elo de ligação entre um ou mais estados origem com um ou mais estados destino. Para que uma transição ocorra é preciso que o(s) estado(s) origem esteja(m) ativo(s) e as restrições em seu rótulo sejam satisfeitas, ou seja, que o evento esperado ocorra e que uma condição de guarda seja satisfeita. Obedecidas estas restrições, ocorre a transição do(s) estado(s) corrente(s) para o(s) estado(s) destino(s) da transição disparada e, durante a transição, é executada uma ação. A definição das informações evento-condição-ação não é obrigatória, uma transição pode até não ter um rótulo. Neste caso ela é denominada transição automática, pois as restrições são consideradas satisfeitas. A forma de representação da transição é uma aresta direcionada interligando a borda do(s) estado(s) origem com a borda do(s) estado(s) destino.

A partir dos elementos identificados na etapa < $A.2.a$ >, alguns deles foram classificados como componentes e outros como atributos conforme agrupamento apresentado na Figura 5.1-i < $A.2.b$ >. Na hierarquia de classes, os estados AND e XOR passam a ser especializações de **bloco** e as transições de **conector** < $A.2.c$ >. Por último, na Figura 5.1-ii estão representados os componentes do Estadograma e

seus respectivos símbolos denominados por rótulos em itálico e apontados por setas tracejadas $< A.2.d >$.

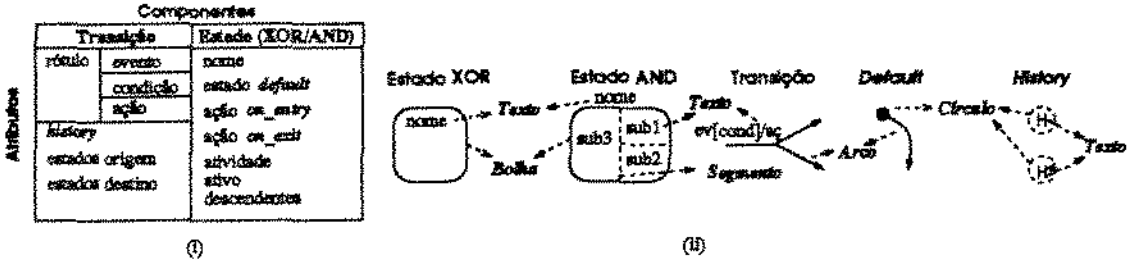


Figura 5.1: (i) Componentes, Atributos e (ii) Símbolos do EdGE.

B) Criar Novas Classes e Incluí-las nas Hierarquias da Plataforma

A Figura 5.2 ilustra as hierarquias da plataforma estendidas pelas classes do exemplo escolhido. As classes em destaque pertencem à plataforma. A classe intermediária **Estado** foi introduzida na hierarquia de **Bloco** visando fatorar as características em comum aos estados AND e XOR.

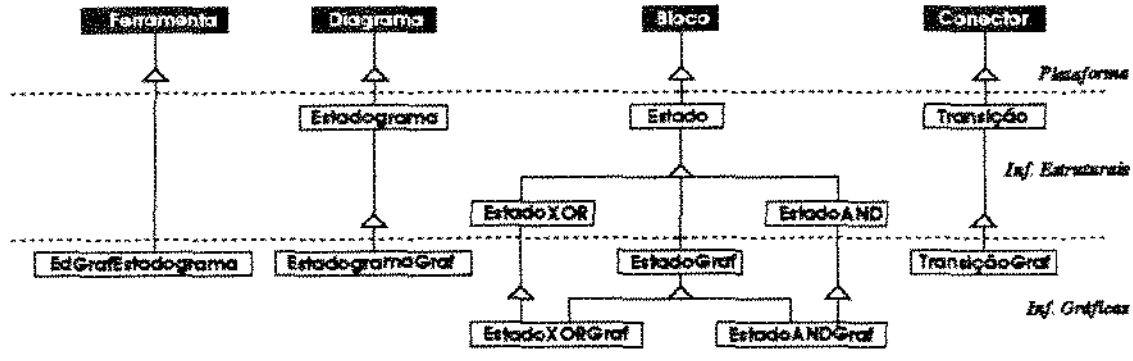


Figura 5.2: Hierarquias da plataforma especializadas para o EdGE.

C) Identificar as Relações entre Classes

Na Figura 5.3 são apresentadas as associações entre as classes criadas na etapa (**B**). Existe uma forte semelhança com as associações estabelecidas a nível de plataforma (Figura 4.2).

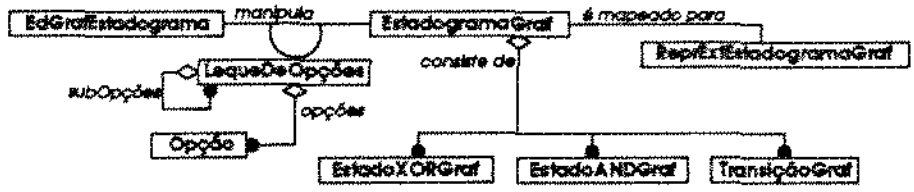


Figura 5.3: Modelo de objetos simplificado do EdGE.

D) Identificar os Atributos

Alguns dos atributos a serem definidos já foram identificados na etapa A, como os atributos dos componentes de Estadograma, além dos símbolos gráficos que os representam. Em relação a Estadograma, os seus componentes devem ser redefinidos. Em vez de ser uma coleção de componentes, passa a ser uma coleção de estados e transições. Como o EdGE é uma ferramenta gráfica, ele precisa de atributos para a sua janela e uma área para desenho de diagramas. Por enquanto, as demais classes não sinalizam a necessidade de novos atributos.

Esta etapa e a próxima já fornecem informações suficientes para construção de uma versão preliminar do diagrama de objetos do EdGE. Entretanto, para evitar a apresentação de muitas informações redundantes, apenas o diagrama de objetos resultante de todo o processo de modelagem é ilustrado ao final do presente capítulo.

E) Identificar as Operações

Analisando as operações definidas nas classes da plataforma e as características a serem acrescentadas, é preciso incluir as operações relevantes ao desenho do diagrama e aquelas que possuem como parâmetro de entrada/saída algum tipo de atributo que foi redefinido na classe especializada. Por exemplo, um **Diagrama** possui uma operação que insere **Componente**, um **Estadograma** deve ter operações para inserção de **Estado** e outra para **Transição**. Todas as operações virtuais das classes da plataforma também devem aparecer redefinidas nas novas classes.

Um conjunto razoável de opções para a edição de estadogramas é ilustrado na árvore da Figura 5.4. A ramificação Arquivo contém as operações de manipulação em arquivo, bem como de finalização da edição. A ramificação Edição contém as operações de criação e alteração dos componentes de um estadograma. A ramificação seguinte, Símbolos, contém operações de manipulação gráfica dos símbolos que compõem os estados, transições e outras representações (por exemplo, símbolo de estado default). A ramificação Extras contém operações que não afetam o diagrama, mas auxiliam na edição. As ramificações são exemplos de instâncias da classe

LequeDeOpções e as opções, exemplos de instâncias da classe **Opção**.

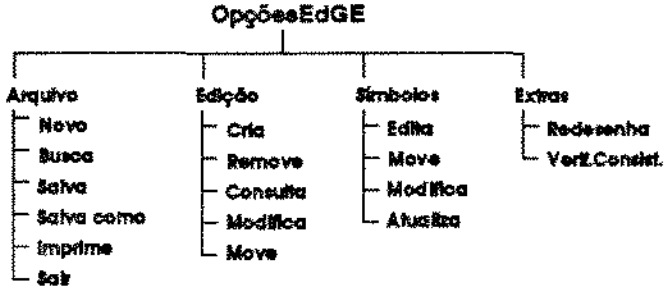


Figura 5.4: Árvore de opções do EdGE.

F & G) Modelagem Dinâmica e Funcional

Uma proposta de leiaute para a janela de EdGE é ilustrada na Figura 5.5. Na região abaixo do nome da janela ficam as opções definidas na etapa E. Na esquerda estão os diferentes tipos de componentes do Estadograma. A seleção de algum deles determina qual componente será criado ao se escolher a opção Edição/Cria. Na modelagem da classe **Transição**, todas as transições são M:N. O caso 1:1 é apenas uma particularização do universo de possibilidades. Entretanto, como o caso 1:1 é mais freqüente ela é caracterizada como uma opção de comportamento, tornando assim a especificação de sua multiplicidade, durante a criação de uma transição dessa natureza, uma tarefa desnecessária. Abaixo dos componentes estão os atributos de configuração para desenho dos símbolos. Como a apresentação do diagrama deve possuir uma configuração fixa, estes atributos não devem estar disponíveis para alteração, mas, para ilustrar alguns dos possíveis atributos, eles estão presentes propositadamente.

Na modelagem dinâmica os cenários de interesse são aqueles para capturar as interações do usuário com os objetos gráficos. Ocorrem durante a execução das funções de manipulação de componentes e símbolos, tais como `cria`, `remove`, `consulta` e `modifica`.

Em seguida, na modelagem funcional as funções que precisam ser detalhadas através de DFDs são: `verificar consistência` (depende das regras sintáticas e semânticas da notação), `busca` e `salva`. Recomendações sobre como modelar estas duas últimas funções estão presentes na Seção 4.3.1. Exemplos de DFDs para as funções de edição (cria estado, modifica transição) são apresentados na Seção 4.1.5.

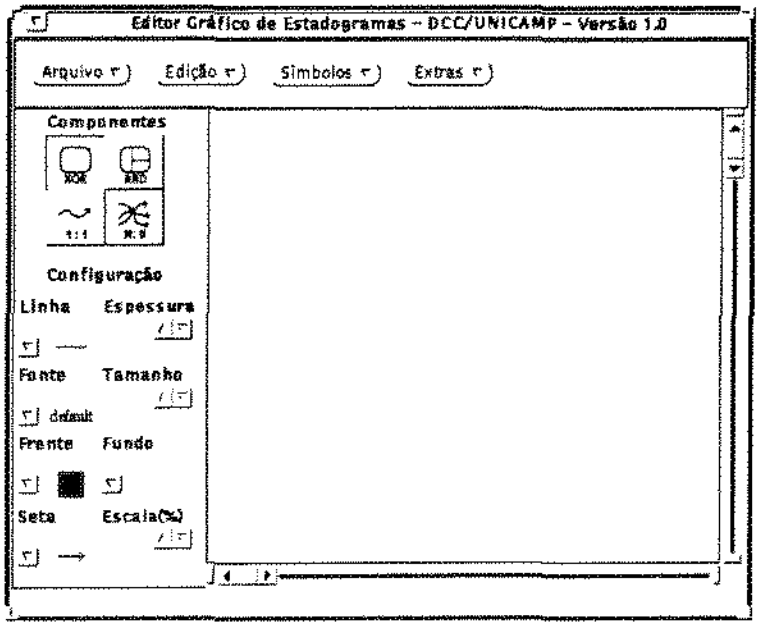


Figura 5.5: Leiaute de uma Janela para o Editor Gráfico de Estadogramas.

5.2.2 Projeto

Os fatores que devem ser considerados no projeto do EdGE são:

memória: os diagramas manipulados não requerem um volume grande de dados para serem armazenados.

facilidade de uso (usabilidade): é importante que o usuário possa manipular com facilidade os recursos do editor.

tempo de execução (eficiência): não é um fator crítico ao sucesso do sistema, pois os processamentos realizados são bastante simples. Apenas algumas operações desempenham atividades mais complexas, tal como a de verificação de consistência e o leiaute automático.

Sistema Gerenciador da Base de Dados: pode ser útil, mas não é necessário. As operações realizadas sobre as bases de dados dos diagramas são leitura e armazenamento de todos os seus dados. Operações como ordenação, busca, seleção, inserção e remoção não são necessárias. Contudo, o uso de um Banco de Dados Orientado a Objetos ou de uma linguagem de programação, que implemente o conceito de objetos persistentes, pode ser conveniente.

recursos auxiliares: estruturas de dados para Árvores e Listas servem para organização dos componentes do diagrama; algoritmos para desenho automático dos diagramas; etc.

A seguir é apresentada a estrutura de dados Tree, que serve para organizar estados de um diagrama. A estrutura List, para organizar as transições, já foi descrita no capítulo anterior.

```
struct Element {el:*< T >; parent:*Element; firstChild:*Element;
rightBrother:*Element; leftBrother:*Element};

struct KeyType {code:Integer; identifier:String; position:Coord; region:Coord[2]};

enum Orientation = {parent, firstChild, rightBrother, leftBrother, leftmostBrother};
```

* Estrutura de Dados: Tree *

classe
Tree< T >
atributos
root:*Element
current:*Element
operações

$create()		/*cria árvore vazia*/
empty():Boolean		/*verifica se árvore está vazia*/
insert(e:*< T >, Orientation):Boolean	where:	/*insere "e" na posição indicada por "where", que pode ser [root/parent/leftmostChild/nextBrother/previousBrother], "current" passa a ser "e", retorna True se sucesso*/
delete(e:*< T >):Boolean		/*remove elto corrente e recursivamente os seus filhos, retorna True se sucesso*/
deleteOnly(e:*< T >):Boolean		/*remove apenas o elto corrente, reorganiza a árvore ajustando a posição dos filhos do elto removido, retorna True se sucesso*/
update(e:*< T >):Boolean		/*troca o elto corrente por "e", retorna True se sucesso*/
retrieve():*< T >		/*retorna o elto corrente, ou Null se $\nexists$*/
goto(where:Orientation): Boolean		/*posiciona elto conforme indicado por "where", neste caso pode ser [root/parent/leftmostChild/nextBrother/previousBrother], retorna True se $\exists$*/
find(key:KeyType):Boolean		/*posiciona elto corrente cuja identificação coincide com "key", retorna True se achar*/

5.2.3 Resultado

A Figura 5.6 contém as especializações das classes **Ferramenta**, **Diagrama** e **Representação Externa**. Algumas diferenças comumente encontradas entre estas classes e suas especializações estão na redefinição das associações herdadas (veja associação entre **Diagrama** e **Representação Externa**) e inclusão de operações de mesmo nome, mas com diferenças no tipo dos parâmetros de entrada/saída, por exemplo, as operações `insere(Componente)`, `insere(Estado)` e `insere(EstadoGraf)`, pertencentes a classes diferentes da hierarquia **Diagrama**. Os atributos `estados` e `estadoRaiz` da classe **Estadograma** fazem referência, respectivamente, à hierarquia de estados e ao estado raiz desta hierarquia. Entre **Estadograma** e **EstadogramaGraf** há uma redefinição dos atributos `estados` e `transições`. Entre **ReprExtEstadograma** e **ReprExtEstadogramaGraf** a diferença está na implementação das suas operações, a semântica permanece a mesma.

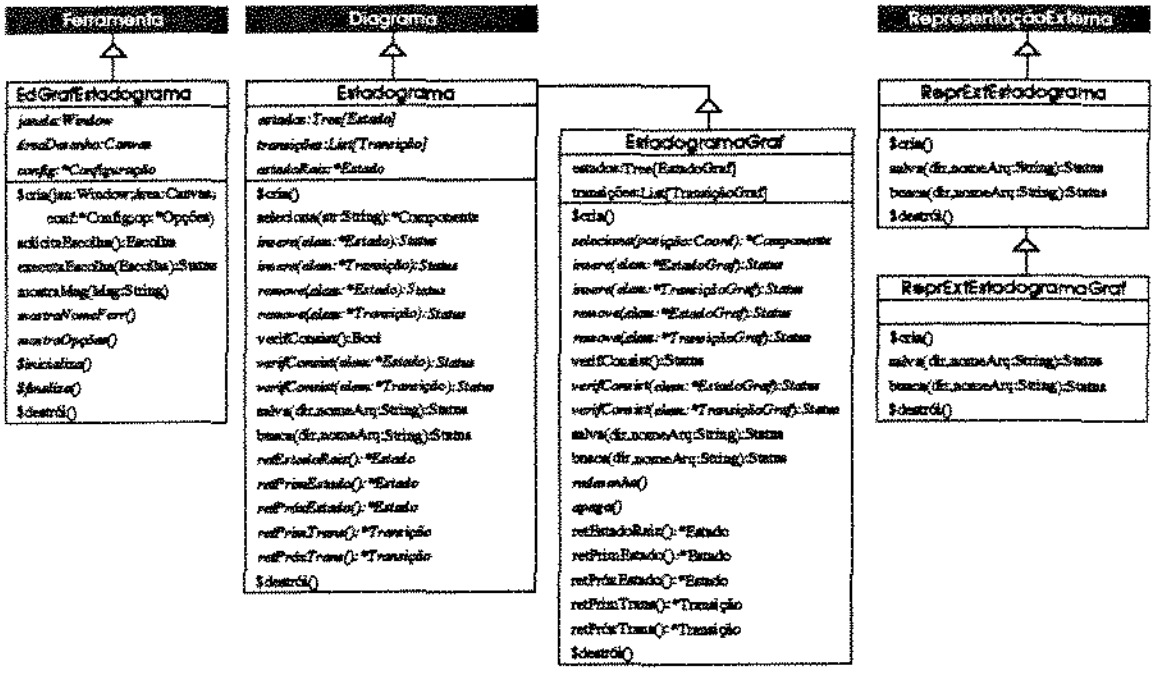


Figura 5.6: Diagrama de objetos para o EdGE.

Os atributos e as operações que aparecem em *itálico* foram incluídos no processo de especialização. Esta representação não faz parte da notação do método OMT, mas ajuda a visualizar as diferenças entre as classes descendentes e ancestrais. Entre **Estadograma** e **EstadogramaGraf** as principais diferenças são os atributos e as operações que foram inseridas para atender os aspectos gráficos do diagrama alvo.

Na Figura 5.7 estão ilustradas as especializações da classe **Componente**. Do mesmo modo que na classe **Estadograma**, os componentes também são divididos em dois grupos: um contendo as informações estruturais (**Estado**, **EstadoXor**, **EstadoAnd** e **Transição**) e outro contendo, adicionalmente, as informações gráficas (**EstadoGraf**, **EstadoXorGraf**, **EstadoAndGraf** e **TransiçãoGraf**). As informações gráficas correspondem aos símbolos ilustrados na Figura 5.1.

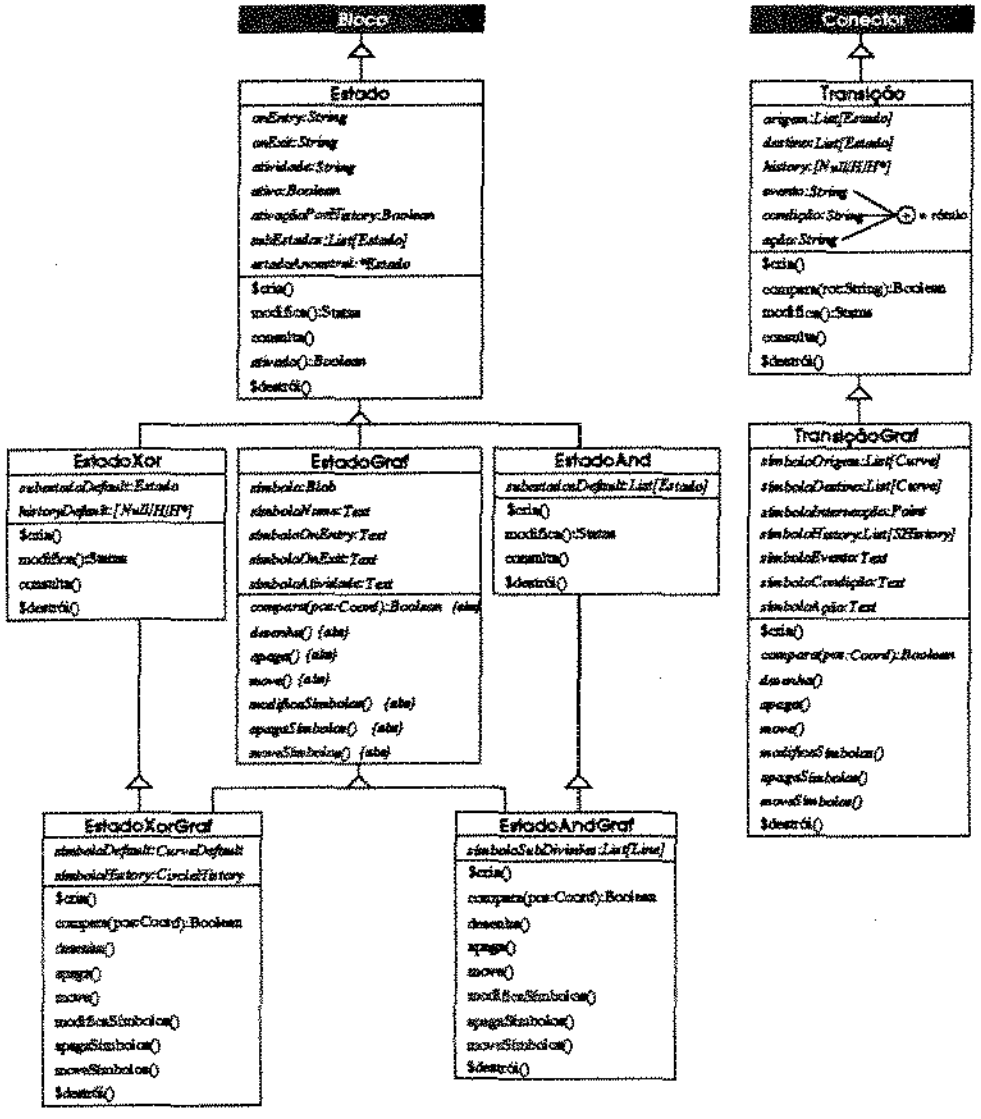


Figura 5.7: Diagrama de objetos para o EdGE (continuação).

Os atributos `onEntry` e `onExit`, da classe **Estado**, indicam as ações que devem ser executadas antes e depois da ativação do estado e o atributo `atividade` indica o

que ocorre durante o período em que o estado estiver ativo. No modelo, o tipo destes atributo referencia apenas o nome da ação ou atividade. Se for preciso, em vez de um nome, o tipo do atributo pode ser um apontador para a função correspondente ou até mesmo uma classe. Uma das diferenças entre **EstadoXOR** e **EstadoAND** é a quantidade de sub-estados *default* que cada um pode ter, no primeiro caso é permitido apenas um, enquanto que no segundo, a quantidade de estados *default* deve ser igual ao número de seus sub-estados.

Supondo que: a representação dos estados do estadograma esteja implementada na forma de "bolhas" aninhadas (Figura B.1), e que se deseja mudar para uma outra representação, por exemplo, uma árvore, onde os *nós* representam os estados e as *arestas* a relação de hierarquia entre eles. Como modelar a nova situação, a partir das hierarquias de classes ilustradas nas Figuras 5.6 e 5.7? A resposta é a seguinte: No mesmo nível hierárquico das classes **EstadoXorGraf** e **EstadoAndGraf**. Uma alternativa é criar bibliotecas de classes semelhantes, uma para cada forma de representação. Escolhida a forma de representação mais adequada, então faz-se a inclusão da biblioteca correspondente à escolha. Em situações mais simples, a solução pode ser apenas especializar uma classe e redefinir a implementação de algumas operações. O fundamental é escolher o nível da hierarquia que seja mais adequado e isto deve ser feito com cautela.

Conforme os modelos apresentados neste capítulo, as extensões de Estadogramas em relação aos Diagramas de Transição de Estados (citadas no Apêndice B) são modeladas como:

- hierarquia: através de atributos que indicam qual é o estado pai (`estadoAncestral`) e quais são os estados filho (`subEstados`);
- ortogonalidade: através da classe **EstadoAnd**. Todos os seus subestados são considerados ortogonais;
- comunicação: mecanismo intrínseco à semântica de Estadogramas. É uma consequência do atributo `ação`, definido no rótulo da classe **Transição**. Por exemplo, uma transição pode ocorrer e o evento definido pela ação pode desencadear outra transição, ou seja, ocorre uma reação em cadeia.

5.2.4 Alternativas

A separação das informações estruturais de gráficas do diagrama alvo permitiu mais de uma alternativa de modelagem. No modelo ilustrado na Figura 5.7, há um emprego

excessivo do mecanismo de herança múltipla. Se isto for um inconveniente, existem outros modelos, os quais serão apresentados a seguir.

O modelo da Figura 5.8 toma como estratégia derivar duas hierarquias paralelas de classes que se relacionam através de uma associação. Cada hierarquia contém uma das categorias de informação (estrutural ou gráfica). Esta alternativa, além de eliminar a necessidade de herança múltipla, facilita a desassociação ou inclusão das informações gráficas, bastando ignorar/destruir ou criar os objetos gráficos, respectivamente. O ponto negativo desta alternativa é a necessidade de se ter um bom controle da consistência entre os dois conjuntos de objetos. Uma alteração estrutural deve ser refletida no objeto gráfico e vice-versa.

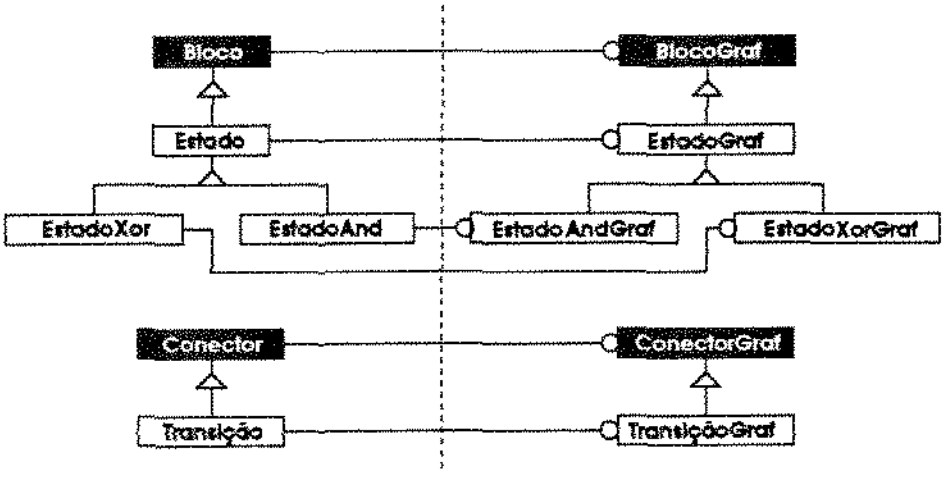


Figura 5.8: Diagrama de Objetos com *hierarquias paralelas*.

O modelo da Figura 5.9 adota a estratégia de intercalar classes das duas categorias de informação através do mecanismo de herança. Ele é uma alternativa mais simples e não apresenta as desvantagens dos modelos anteriores como excesso de herança múltipla e manutenção de consistência entre objetos. Entretanto, a separação das duas categorias de informação é apenas simbólica, ou seja, as características gráficas definidas em níveis hierárquicos superiores são transmitidas para os seus descendentes.

Dentre as alternativas apresentadas, a melhor solução pode depender também do contexto em que as classes estarão inseridas, tais como, a linguagem de programação utilizada (permite herança múltipla?), a natureza da ferramenta (gráfica ou não) e outros fatores.

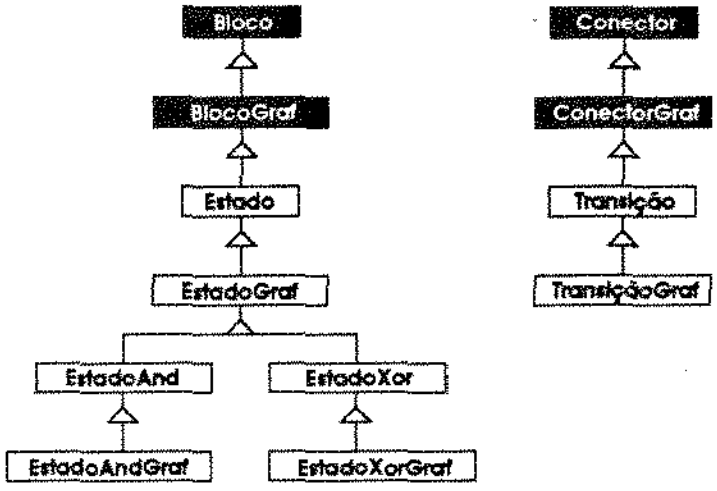


Figura 5.9: Diagrama de Objetos *intercalado*.

5.3 Comentários

Este capítulo procurou dar uma visão mais concreta do emprego da plataforma na modelagem das FBDs. A atividade descrita neste capítulo foi importante no sentido de que algumas deficiências puderam ser detectadas e corrigidas. A referência básica para este capítulo é o conteúdo do capítulo anterior e os artigos sobre estadogramas, principalmente [Har87a] e [HPSS87]. No âmbito do Projeto Xchart, em trabalho futuro, deverá ser feito um editor gráfico para Xchart.

Capítulo 6

Conclusão

Como considerações finais do presente trabalho, este capítulo apresenta os problemas enfrentados em sua realização, faz uma auto-crítica, destaca as contribuições e sugere extensões que possam dar continuidade ao que foi desenvolvido.

Alguns fatos importantes constituem os principais temas abordados pela plataforma:

1. **diagramas** são largamente usados na representação de modelos abstratos das aplicações em Computação;
2. o uso de **ferramentas automatizadas** tem sido cada vez mais necessário para agilizar o processo de desenvolvimento de software;
3. extensibilidade e reusabilidade são propriedades bastante desejáveis aos produtos de software. O **paradigma de objetos** vem sendo considerado uma alternativa promissora para alcançar estas duas propriedades.

Estes fatos enfatizam a relevância do contexto no qual a plataforma está inserida.

6.1 Problemas Encontrados

Não é trivial analisar o domínio de um problema quando não se observa uma uniformidade nas amostras. Nas FBDs, os aspectos comportamental e funcional apresentam poucos elementos em comum. Apenas o aspecto estrutural é o mais uniforme e, portanto, pôde ser adequadamente capturado através do MO. Diante desta situação, a preocupação foi encontrar um meio termo para o enfoque dado aos modelos. Se for

muito genérico, o esforço de especialização da plataforma pode ser muito grande. Por outro lado, se for muito específico, acaba por limitar demasiadamente o escopo de utilização.

O método OMT foi muito importante para tornar o processo de modelagem mais organizado e menos empírico. Contudo, os benefícios de orientação a objetos e do método só são adequadamente explorados após um esforço inicial de aprendizado. Uma experiência anterior no emprego do método OMT seria muito útil para tornar o processo de modelagem mais eficiente. Por exemplo, sabendo-se previamente **como** e **quando** as decisões iniciais seriam usadas posteriormente ou ainda, reconhecendo quais os passos desnecessários são formas de otimização dos esforços.

Mesmo não sendo essencial, é muito importante ter ferramentas que auxiliem o uso do método OMT. Um exemplo é o StP/OMT - *Software through Pictures/Object Modeling Technique*. O StP/OMT facilita as atividades de edição dos diagramas, impressão, verificação de consistência de/entre modelos e geração parcial de código (C++, Ada, Smalltalk) do Modelo de Objetos. Infelizmente, não foi possível ter acesso a ferramentas deste tipo durante a realização do presente trabalho. Mesmo assim, tal recurso não elimina a necessidade de conhecer o paradigma de objetos e o processo de modelagem do método que é dado suporte.

6.2 Análise Crítica do Trabalho

Como observações ao presente trabalho, dois aspectos devem ser mencionados. Um deles é com relação ao emprego do método OMT e o outro, ao produto resultante da modelagem.

No primeiro aspecto, é possível que alguns passos do processo tenham sido interpretados e aplicados de forma diferente do proposto por Rumbaugh et al. Possivelmente, algumas destas diferenças ocorreram por causa da existência de pontos obscuros no método, tais como os comentados na Seção 3.2.2.

No outro aspecto, a intenção era apresentar dados numéricos ou qualitativos de comparação entre o produto resultante da modelagem (a plataforma) e o objetivo estabelecido na Seção 1.4. Contudo, não foi possível avaliar se a extensibilidade e a capacidade de reutilização das FBDs, construídas com o auxílio da plataforma, são as esperadas, pois são medidas que dependem da implementação de várias ferramentas (do tipo FBD). Conforme Prieto [PD90], trabalho deste tipo, envolvendo Análise de Domínio, passam por um processo de evolução que requer tempo, experiência e acúmulo de conhecimento (mais detalhes encontram-se na Seção 1.6). Neste sentido, o Projeto Xchart deverá ter uma participação fundamental, a partir do momento em

a plataforma estiver sendo usada efetivamente.

6.3 Contribuições

O Projeto Xchart foi o estímulo para a realização do presente trabalho. A princípio, ele será beneficiado com o emprego da plataforma na construção de um ambiente flexível de ferramentas (veja Seção 1.5). Por não ser tão específica ou dar privilégios a uma notação em particular, a plataforma também pode ser empregada em outros projetos.

Aplicações em Computação, quando modeladas corretamente, através do paradigma de objetos, apresentam muitas vantagens (Seção 3). Porém, quando os objetos e demais aspectos envolvidos não são bem definidos, conseqüências negativas podem afetar todo o sistema - alguns programadores assumem erroneamente que apenas usando a linguagem C++, já estarão fazendo uso do paradigma de objetos. Visando minimizar a ocorrência de problemas de tal natureza, a plataforma serve como um *template* para a modelagem das FBDs e o procedimento descrito no Capítulo 5, como um guia para o seu desenvolvimento. Acredita-se que a simplicidade dos conceitos definidos na plataforma contribuam para que ela seja de fácil entendimento e utilização.

6.4 Temas para Futuras Pesquisas

O uso da plataforma no Projeto Xchart será um importante laboratório para o seu aprimoramento. Efetivar as melhorias através da coleta e análise das sugestões fica como uma proposta para trabalhos futuros.

O enfoque dado à plataforma é mais voltado à parte estrutural das FBDs, principalmente ao aspecto do diagrama. Uma sugestão é realizar uma abordagem mais algorítmica à hierarquia **Ferramenta**, ou seja, enfocar a parte de implementação dessas classes. Pode-se observar na Figura 5.2, que existe um espaço vago entre o topo da hierarquia **Ferramenta** e a sua especialização **EdGrafEstadograma**. Este espaço vago pode ser preenchido por classes como editores gráficos, geradores automáticos de leiaute e simuladores, todos genéricos. Implementar tais classes é uma atividade não trivial e envolve estudos de algoritmos que atendam vários casos particulares de diagrama. Por exemplo, para um editor gráfico de propósito geral devem ser estudados algoritmos que, a partir de regras de composição do diagrama ou de parâmetros, permitam compor os elementos do diagrama consistentemente.

É importante ter implementado também algoritmos para desenho automático de diagramas. Alguns são bastante conhecidos, tais como o de *Sugiyama* [STT81], que procura minimizar o cruzamento de arestas, *Robins* [Rob87], para grafos dirigidos, *Woods* [Woo81], para grafos não dirigidos e *Reingold* [RT81], para grafos acíclicos. Uma coletânea de trabalhos para desenho de grafos encontra-se em [ET88]. Outros algoritmos também precisam ser analisados, pois muitos não se aplicam quando o tamanho dos nós é variável, situação comum ao contexto de diagramas como Estadogramas e Xchart.

Outra proposta é a implementação de bibliotecas de símbolos gráficos nos vários toolkits que suportam padrões de interface como Motif, Openlook, Windows e Macintosh. Esta é uma forma de aumentar a portabilidade das FBDs.

Apêndice A

Notação OMT

Este apêndice apresenta as notações usadas para representar os modelos da plataforma, segundo o método OMT.

A Figura A.1 apresenta a notação do Modelo de Objetos, chamada Diagrama de Objetos. Nele as classes são representadas por retângulos (*i*), as instâncias de classes por "bolhas" (*ii*) e os relacionamentos por segmentos de reta conectados às classes. O três tipos de relacionamentos usados são: associação (*iii*), especialização da classe 2 a partir da classe 1 (*iv*) e agregação (*v*). Associações com atributos são representadas conforme (*vi*). Relacionamentos também podem ter uma multiplicidade (*viii*).

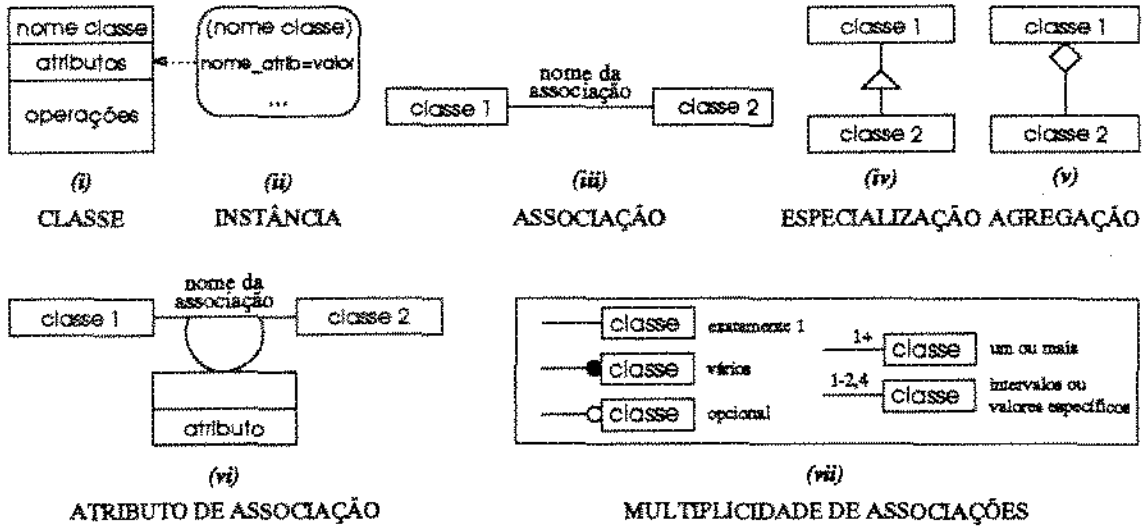


Figura A.1: Notação do Modelo de Objetos.

A Figura A.2 apresenta a notação do Modelo Dinâmico, denominada Estado-

grama. Sua descrição é dada em mais detalhes no Apêndice B. Os estados são representados por "bolhas" (*i*), contendo internamente o nome da atividade executada ao se ativar o estado e das ações (*entry, event e exit*). Os subdiagramas contendo estados concorrentes são representados conforme (*ii*). As transições correspondem a segmentos de reta rotulados (*iii*). O início e fim de seqüências específicas de estados são indicadas por símbolos especiais (*iv*).

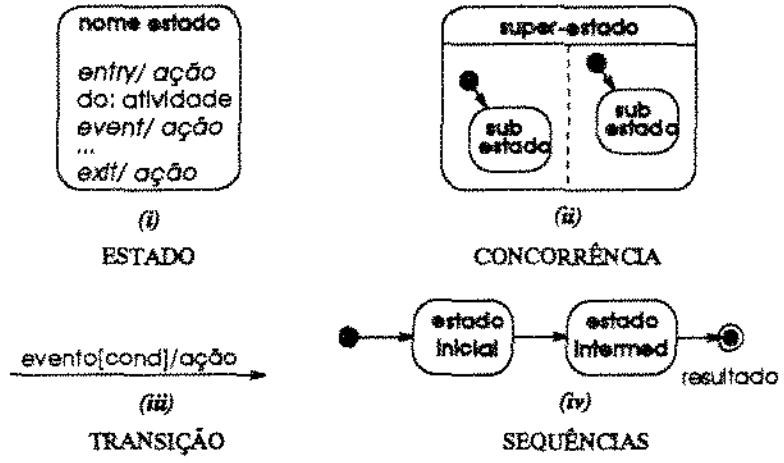


Figura A.2: Notação do Modelo Dinâmico.

A Figura A.3 apresenta a notação do Modelo Funcional, chamada Diagrama de Fluxo de Dados. Os processos (*i*) são representados por elipses, os atores (ou entidades externas) (*ii*) por retângulos, os depósitos de dados (*iii*) por barras paralelas e os fluxos de dados por segmentos de reta. Quando o destino de um fluxo de dados é um depósito de dados, então ele é representado por uma seta especial (*iv*).

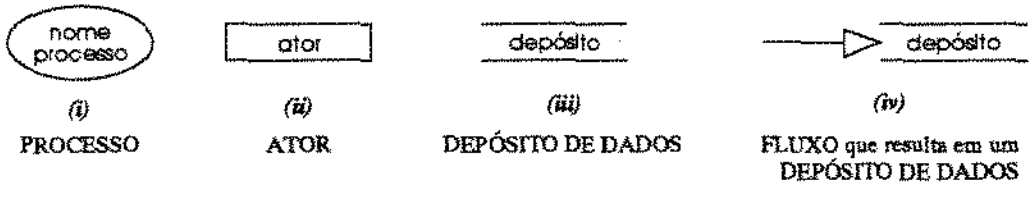


Figura A.3: Notação do Modelo Funcional.

Apêndice B

Estadogramas

Estadograma é uma notação gráfica que estende os Diagramas de Transição de Estados fazendo uso de *hierarquia*, *ortogonalidade* e *comunicação*. É composto por *arestas* orientadas (ou transições) que interligam *estados*. Os estados são representados por retângulos com cantos arredondados e as arestas por arcos dirigidos.

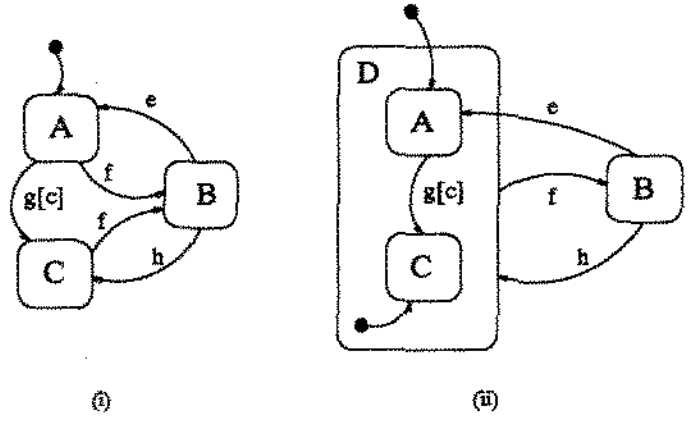


Figura B.1: Exemplo de hierarquia.

A idéia de *hierarquia* é ilustrada na Figura B.1-(ii), onde (ii) e (i) são equivalentes. Os símbolos e, f, g e h são eventos que disparam transições e c (entre colchetes) é uma condição de guarda. Assim, a transição de **A** para **C** ocorre se o estado **A** estiver ativo, o evento g ocorrer e a condição c for verdadeira. A aresta f, que deixa o estado **D** (em (ii)), aplica-se a **A** e a **C**, como em (i). Arestas *default* indicam qual o sub-estado que deve ser ativado. Em (ii), o estado *default* de **D** é o estado **C**. São representadas por uma aresta com um círculo em uma das extremidades. A ativação por *default* é ignorada quando existir uma transição determinando outra situação,

por exemplo, se o evento e ocorrer e o estado **B** estiver ativo, então o sub-estado de **D** que deve ser ativado é o **A**.

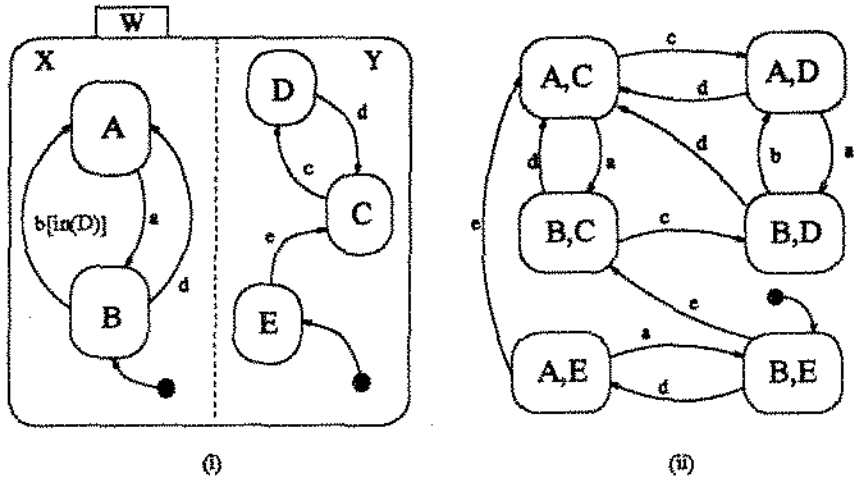


Figura B.2: Exemplo de ortogonalidade.

A *ortogonalidade* em Estadogramas é ilustrada na Figura B.2. Novamente, (*i*) e (*ii*) são equivalentes. O estado **W** consiste dos componentes ortogonais **X** e **Y**. Se **W** estiver ativo, significa que **X** e **Y** também estarão ativos simultaneamente. Ortogonalidade tem a vantagem de evitar o crescimento exponencial do número de estados. O estado **W**, com componentes ortogonais, é considerado um estado do tipo AND, os demais são do tipo XOR.

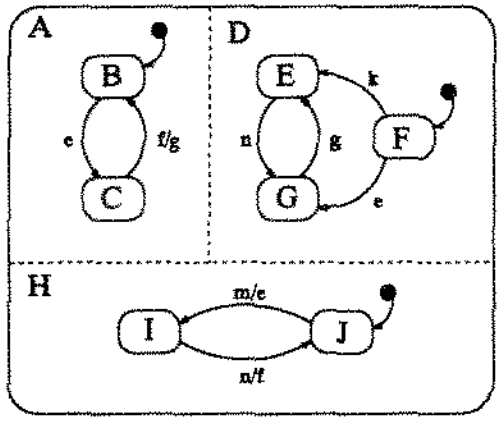


Figura B.3: Exemplo de comunicação.

A Figura B.3 apresenta um exemplo de *comunicação*. Se a configuração corrente for (**B**,**F**,**J**) e o evento *m* ocorrer, a próxima configuração será (**C**,**G**,**I**) em virtude

do evento *e* ser gerado pela transição de **J** para **I** e ter disparado duas transições nas componentes concorrentes **A** e **D**. Este efeito é chamado de reação em cadeia.

Outro mecanismo presente em Estadogramas é o de *history*. Ele serve para "lembrar" de um estado previamente visitado. Na Figura B.4 há uma entrada no estado **X** por *history*. Ao ocorrer o evento *e* e o estado corrente for **K**, então o próximo estado a ser ativado será o mais recentemente visitado entre **A** e **B**. No caso de ser a primeira vez em que o estado **X** estiver sendo ativado, então segue-se a aresta *default*, que conduz ao estado **A**.

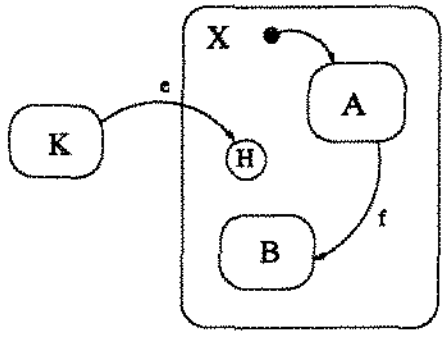


Figura B.4: Exemplo de *history*.

Bibliografia

[Bar86] Paul S. Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[Bat91] J. E. S. Batista. Um Editor Gráfico para Statecharts. MSc Thesis, ICMSC-USP, São Carlos, 1991.

[BBJS92] B. Bruegge, J. Blythe, J. Jackson, and J. Shufelt. Object-Oriented System Modeling with OMT. In *OOPSLA*, pages 359–376. ACM SIGPLAN, 1992.

[BE94] T. Bryant and A. Evans. OO Oversold - Those Objects of Obscure Desire. *Information and Software Technology*, 36(1):35–42, 1994.

[BEP80] L. A. Belady, C. J. Evangelisti, and L. R. Power. Greenprint: A Graphic Representation of Structured Programs. *IBM System Journal*, 19(4):542–553, 1980.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin-Cummings, 1991.

[Bot86] D. Botting. Into the Fourth Dimension: An Introduction to Dynamic Analysis and Design. *Software Engineering Notes*, 11(2):36–48, April 1986.

[Bra77] J. M. Brady. *The Theory of Computer Science: A Programming Approach.* Chapman and Hall, London, 1977.

[Bro83] E. J. Brown. On the Application of Rothon Diagrams to Data Abstraction. *SIGPLAN Notices*, 18(12):17–24, December 1983.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Can93] J. W. L. Cangussu. Execução Programada de Statecharts. MSc Thesis, ICMSC-USP, São Carlos, 1993.

[CB91] G. Caldiera and V. Basili. Identifying and Qualifying Reusable Software Components. *Computer*, 24(2):61–70, February 1991.

[CG75] S. Caine and E. Gordon. PDL - A Tool for Software Design. In AFIPS Press, editor, *Proceedings of the AFIPS National Computer Conference*, volume 44, pages 271–276, Montvale, N.J., 1975. Anahein, CA.

[Chv83] V. Chvalovsky. Decision Tables. *Software Practices & Experiences*, 13:423–429, 1983.

[Chy84] S. C. Chyou. Structure Charts and Program Correctness Proofs. In *Proceedings of the 7th International Conference on Software Engineering*, pages 486–498, 1984.

[CJCC93] R. S. Cabral, S.A. Jansen, P. P. Carreiro, and J. B. Castro. Utilização da Metodologia OMT na Construção de Ferramentas CASE. *VII Simpósio Brasileiro de Engenharia de Software*, pages 268–281, October 1993.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall - Englewood Cliffs, N.j., 1991.

[Dav78] A. Davis. Requirements Language Processing for the Effective Testing of Real-Time Software. *ACM Software Engineering Notes*, 3(5):61–66, November 1978.

[Dav88] Alan M. Davis. A Comparison of Techniques for the Specification of External System Behavior. *Communications of the ACM*, 31(9):1098–1115, September 1988.

[DT72] R. Doran and G. Tate. An Approach to Structured Programming. Technical report, Masey University, June 1972.

[DV77] C. Davis and C. Vick. The Software Development System. *IEEE Transactions on Software Engineering*, 3(1):69–84, January 1977.

[ET88] P. Eades and R. Tamassia. Algorithms for Automatic Graph Drawing: An Annotated Bibliograph. Technical report, University of Queensland, Dept, of Computer Science, St. Lucia, Queensland, 4067, Australia, 1988.

[Fer78] O. Ferstl. Flowcharting by Stepwise Refinement. *SIGPLAN Notices*, 13(1):34–42, January 1978.

[Fil91] Antônio Gonçalves Figueiredo Filho. Um Processo de Síntese de Sistemas Reativos. Master's thesis, DCC-IMECC-UNICAMP, Dezembro 1991.

[FK92] R. G. Fichman and C.F. Kemerer. Object-Oriented and Conventional Analysis and Design Methodologies - Comparison and Critique. *IEEE Computer*, pages 22–39, October 1992.

[FKHT81] Y. Futamura, T. Kawai, H. Horikoshi, and M. Tsutsumi. Development of Computer Program by Problem Analysis (PAD). In *Proceedings of 5th International Conference on Software Engineering*, pages 325–332, 1981.

[FM92] R. Furuuti and P. C. Masiero. Extensões do Ambiente StatSim para Permitir a Simulação de Micro-Passos. Relatório CNPq, ICMSC-USP, São Carlos, 1992.

[For91] R. P. M. Fortes. Uma Ferramenta de Apoio à Utilização de Statecharts para Especificação do Comportamento de Sistemas de Tempo Real Complexos. MSc Thesis, ICMSC-USP, São Carlos, 1991.

[Gro77] P. Grouse. Flowblocks - A Technique for Structured Programming. *SIGPLAN Notices*, pages 46–56, 1977.

[Har87a] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har87b] S. Harrington. *Computer Graphics: A Programming Approach.* McGraw-Hill, 1987.

[Hea90] J. Hess and et al. A Domain Analysis Bibliography. Technical Report CMU/SEI-90-SR-3, SEI-Carnegie Mellon University, Pittsburgh, PA, 1990.
http://www.sei.cmu.edu/products/publications/90.reports/90.sr.003.html.

[HH93] D. Hix and H. R. Hartson. *Developing User Interfaces: Ensuring Usability through product and process.* John Wiles & Sons, Inc., 1993. ISBN 0471-57813-4.

[HLN+90a] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, April 1990.

[HLN+90b] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, April 1990.

[HNRT79] D. Harel, P. Norvig, J. Rood, and T. To. A Universal Flowcharter. In *Proceedings of AIAA Computers in Aerospace Conference*, pages 218–224, 1979.

[HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. Technical Report, The Weizmann Institute of Science — Department of Applied Mathematics, Rehovot 76100, Israel, 1985.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. *Proceedings of 2nd. IEEE Symposium on Logig in Computer Science*, pages 54–64, 1987.

[HS80] S. Hanata and T. Satoh. Compact Chart - A Program Logic Notation with High Descibability and Understand-ability. *SIGPLAN Notices*, 15(9):32–38, September 1980.

[iLI89] i Logix Inc. STATEMATE: Semantics of Statecharts, August 1989. Burlington, MA.

[Jac75] M. A. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addisson-Wesley, Reading, MA, 1992.

[Jon87] D. Jonsson. Pancode and Boxcharts: Structured Programming Revisited. *SIGPLAN Notices*, 22(8):89–98, August 1987.

[Jon89] Dan Jonsson. Graphical Program Notations: On Tripp's Survey, the Past, and the Future. *ACM SIGSOFT Software Engineering Notes*, 14(5):78–79, July 1989.

[JT79] R. W. Jensen and C. C. Tonies. *Software Engineering.* Prentice-Hall, Englewood Cliffs, N.J., 1979. pp. 267-273.

[KC93] G. M. Karam and R. S. Casselman. A Cataloging Framework for Software Development Methods. *IEEE Computer*, pages 34–45, February 1993.

[KS80] Y. Kanda and M. Sugimoto. Software Diagram Description: SDD and Its Application. In *Proceedings of COMPSAC-80*, pages 300–305, 1980.

[LHL96] Fábio N. Lucena, Mário Massato Harada, and Hans K.E. Liesenberg. Semântica Formal de Xchart. *Em preparação*, 1996. `http://www.dcc.unicamp.br/projects/Xchart/`.

[Lie93] Hans Liesenberg. Projeto Xchart. Versão preliminar, DCC-IMECC-UNICAMP, 1993. `http://www.dcc.unicamp.br/projects/Xchart/`.

[Lin77] G. H. Lindsey. Structure Charts: A Structured Alternative to Flowcharts. *SIGPLAN Notices*, 12(11), November 1977.

[LL94] Fábio N. Lucena and Hans K. E. Liesenberg. A Statechart Engine to Support Implementations of Complex Behaviour. *XXI Semish*, pages 177–191, 1994. Caxambu/MG, Brazil.

[LL96] Fábio N. Lucena and Hans K.E. Liesenberg. Estendendo Statecharts: Uma Primeira Proposta. *Em preparação*, 1996. `http://www.dcc.unicamp.br/projects/Xchart/publicacoes`.

[LLB95] Fábio N. Lucena, Hans K. E. Liesenberg, and Luiz E. Buzato. Xchart-Based Complex Dialogue Development. In *Simpósio Nipo-Brasileiro de Ciência e Tecnologia*, pages 387–396, Campos do Jordão/SP, August 1995.

[Luc93] Fábio N. Lucena. Construção de Interface Homem-Computador – O uso de Estadogramas na especificação e implementação de interfaces. Master's thesis, DCC-IMECC-UNICAMP, Março 1993.

[Mar79] D. Marca. A Method for Specifying Structure Programs. *ACM SIGSOFT Software Engineering Notes*, 4(3):22–31, July 1979.

[McC85] R. McCain. Reusable software component construction: A product oriented paradigm. pages 125–135, Long Beach, CA, October 21-23 1985. 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference.

[MM85] James Martin and Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall, Inc., 1985.

[Mor82] B. Moret. Decision Trees and Diagrams. *ACM Comput. Surv.*, 14(4):593–623, December 1982.

[MP92] David E. Monarchi and Gretchen I. Puhr. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, 35(9):35–47, September 1992.

[Nei81] J. Neighbors. *Software Construction Using Components.* PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 1981.

[NS73] I. Nassi and B. Schneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Notices*, 8(8):12–26, August 1973.

[NS87] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* 1987.

[Pag87] F. G. Pagan. A Graphical FP Language. *SIGPLAN Notices*, 22(3):21–39, March 1987.

[PD90] R. Prieto-Díaz. Domain Analysis: An Introduction. *ACM Software Engineering Notes*, 15(2):47–54, April 1990.

[Pet77] J. Peterson. Petri-nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[PT90] Frances N. Paulisch and Walter F. Tichy. EDGE: An Extendible Graph Editor. *Software - Practice and Experience*, 20(S1):S1/63–S1/88, June 1990.

[Rae85] Georg Raeder. A Survey of Current Graphical Programming Techniques. *IEEE Computer*, pages 11–25, August 1985.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Rob81] P. N. Robillard. On an Experimental Tool for General Software Development. In *Proceedings of the 14th Hawaii International Conference on System Sciences*, pages 90–97, 1981.

[Rob87] G. Robins. The ISI Grapher: A Portable Tool for Displaying Graphs Pictorially. *Symboliikka '87 Helsinki, Finland*, August 1987. Information Sciences Institute, Marina Del Rey, CA.

[RS82] A. Rockstrom and R. Saracco. SDL - CCITT Specification and Description Language. *IEEE Trans. Commun.*, 30(6):1310–1318, June 1982.

[RT81] E. M. Reingold and J. S. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[Sil94] Maria Inês Vale Silva. Desenho Automático de Diagramas. MSc Thesis, DCC-UNICAMP, 1994.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, 1981.

[TCU86] R. P. Taylor, N. Cunniff, and M. Uchiyama. Learning, Research and the Graphical Representation to Programming. In *Proceedings of 1986 Fall Joint Computer Conference*, pages 56–63, 1986.

[TM91] R. Tutumi and P. C. Masiero. Simulação Visual de Statecharts. In *X Congresso de Iniciação Científica e Tecnológica em Engenharia*, page 669, São Carlos, 1991. CICTE-EESC.

[Tri88] Leonard L. Tripp. A Survey of Graphical Notations for Program Design - An Update. *ACM SIGSOFT Software Engineering Notes*, 13(4):39–44, 1988.

[VL90] John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Buiding Domain-Specific Graphical Editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[Wal92] Ian J. Walker. Requirements of an Object-Oriented Design Method. *Software Engineering Journal*, 7(2):102–113, March 1992.

[War74] J. D. Warnier. *Logical Construction of Programs*. Paris, 1974.

[War86] P. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, 12(2):198–210, February 1986.

[WBWW90] R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall - Englewood Cliffs, N.J., 1990.

[Wit77] R. W. Witty. The Design and Construction of Hierarchically Structured Software. In *Proceedings of Pragmatic and Sensible Software*, pages 361–388, February 1977.

[Woo81] D. R. Woods. *Drawing Planar Graphs.* Ph.D. Thesis, Report No. STAN-CS-82-943, Stanford University, Department of Computer Science, June 1981.

[YC78] E. Yourdon and L. Constantine. *Structured Design.* Yourdon Press, New York, 1978. pp. 42-50.

[Zav82] P. Zave. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, May 1982.

Índice

P

R

S

T

X