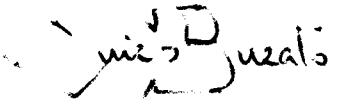
# Um Editor Gráfico para Stabilis/Vigil

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rodrigo Netto Lacerda e aprovada pela Banca Examinadora.

Campinas, 3 de dezembro de 1997.



Prof. Dr. Luiz Eduardo Buzato
(Orientador)

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

Instituto de Computação
Universidade Estadual de Campinas

# Um Editor Gráfico para Stabilis/Vigil

**Rodrigo Netto Lacerda**

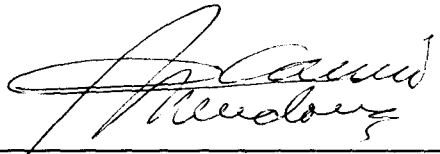
Dezembro de 1997

**Banca Examinadora:**

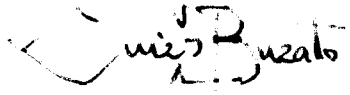
- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Profa. Dra. Clarisse Sieckenius de Souza
  (Departamento de Informática—PUC/Rio)
- Prof. Dr. Candido Ferreira Xavier de Mendonça Neto
  (Instituto de Computação—UNICAMP)
- Prof. Dr. Hans Kurt E. Liesenberg (Suplente)
  (Instituto de Computação—UNICAMP)

Tese de Mestrado defendida e aprovada em 28 de novembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Profª. Drª. Clarisse Sieckenius de Souza



Prof. Dr. Cândido Ferreira Xavier de Mendonça Neto



Prof. Dr. Luiz Eduardo Buzato

# Agradecimentos

Primeiramente, gostaria de agradecer aos meus pais, Renato e Marly, e a todos meus irmãos, Mauro Henrique, Vânia, Eliane e Eugênio, que sempre me deram apoio para que eu conseguisse atingir este objetivo. Em especial, gostaria de agradecer a minha namorada, Cristiane, pela força e paciência, apesar da distância que nos separa.

Gostaria de expressar minha profunda gratidão aos meu amigos e companheiros da Toca do Queijo (Célio, Daniel, Elbson e Mauro), pelo convívio e pela ajuda durante estes anos aqui em Campinas. Gostaria de agradecer também ao meu grande amigo Doriguetto, aos alunos, professores e funcionários do Instituto de Computação, enfim, a todos aqueles que direta ou indiretamente ajudaram na concretização deste trabalho.

E, finalmente, gostaria de agradecer ao professor Buzato pela orientação e aos órgãos de fomento à pesquisa CNPq, através do processo número 830231/95-1, e FAPESP, através do processo número 96/0156-3, pelo apoio financeiro, sem os quais a execução deste trabalho não seria possível.

# Resumo

O objetivo deste trabalho é o desenvolvimento de um conjunto de mecanismos de modo a automatizar a captura, a geração parcial de código, o traçado e o posicionamento dos elementos gráficos de modelos de objetos. Este conjunto de mecanismos, integrados em uma única ferramenta, denominada editor gráfico, deve prover uma interface de programação visual de alto nível para o ambiente de programação distribuída formado por Stabilis/Vigil.

# Abstract

The goal of this work is the development of a set of mechanisms to automate the capture, code generation, routing and positioning of graphical elements of object models. This set of mechanisms have been integrated into a graphical that provides a high-level visual programming interface for the distributed programming environment Stabilis/Vigil.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Lista de Programas

# Introdução

## Motivação

O desenvolvimento de aplicações orientadas a objetos tem sido, em geral, apoiado por metodologias de projeto orientado a objetos. Quando isto acontece, essas aplicações podem ser representadas por modelos abstratos que capturam a sua estrutura e o seu comportamento. Esses modelos abstratos são normalmente denominados modelos de objetos e, na maior parte das vezes, são representados através de notações gráficas, com vantagens para a visualização e a compreensão da funcionalidade das aplicações modeladas. Entretanto, a manipulação de representações gráficas de modelos de objetos de aplicações maiores (várias dezenas de classes e relacionamentos) somente é viável se for automatizada. Neste contexto, o objetivo deste trabalho é o desenvolvimento de uma ferramenta para a edição, captura, traçado e posicionamento dos elementos gráficos de modelos de objetos e a geração automática de código fonte a partir do modelo capturado. Este conjunto de mecanismos, integrados em uma única ferramenta, denominada editor gráfico, deve prover uma interface de programação visual de alto nível para o ambiente de programação distribuída formado por Stabilis/Vigil. Stabilis/Vigil provê um ambiente de programação que auxilia o desenvolvimento de aplicações tolerantes a falhas, distribuídas, orientadas a objetos e estruturadas utilizando ações atômicas distribuídas. Ações atômicas fornecem mecanismos para tolerância a falhas de *hardware*. A comunicação entre objetos distribuídos é realizada através de chamada de procedimento remoto.

A integração do editor gráfico ao ambiente de programação permite a geração automática de parte do código fonte de programas distribuídos tolerantes a falhas, reduzindo o esforço de projeto, implementação e gerência desses programas.

## Organização

No Capítulo 1, introduzem-se os conceitos básicos que fundamentam o desenvolvimento deste trabalho. O objetivo deste Capítulo é dar subsídios para o entendimento dos capítulos restantes. No Capítulo 2, é apresentada uma visão geral da arquitetura de

*software* do ambiente de programação distribuída, inclusive a do editor gráfico. Nos Capítulo 3 e 4, são apresentados detalhadamente os componentes dos módulos Interface e Aplicação que constituem a arquitetura de *software* do editor gráfico. No Capítulo 5, é apresentado um exemplo do uso do editor gráfico no desenvolvimento de programas distribuídos orientados a objetos. O Capítulo 6 é dedicado à apresentação de trabalhos relacionados. Finalmente, no Capítulo 7 é feito uma análise e conclusões sobre o projeto e implementação do editor gráfico, apontado contribuições e áreas de estudo que poderão ser exploradas no futuro.

# Capítulo 1

# Fundamentos

A construção de um editor gráfico para um ambiente de programação distribuída requer o estudo de tópicos de várias áreas da ciência da computação. Por se tratar de um sistema interativo, o editor gráfico desenvolvido neste trabalho, além de abordar naturalmente a área de sistemas de interação homem-computador, também aborda as seguintes áreas: Desenho de Grafos, Orientação a Objetos e Distribuição. A primeira se deve às facilidades de leiaute automático implementadas e, as restantes, às características inerentes do ambiente de programação distribuída Stabilis/Vigil.

Dada esta diversificação, optou-se por apresentar neste Capítulo uma breve explanação de conceitos envolvidos em cada uma dessas áreas e que possuam relacionamento direto com o presente trabalho. Os conceitos aqui apresentados visam dar os fundamentos necessários para a apresentação dos Capítulos restantes.

## 1.1 Sistemas Interativos

Por *interação homem-computador* entende-se tudo o que ocorre entre um ser humano (usuário) e um computador. Um *sistema interativo* é utilizado pelo usuário para a realização de vários tipos de tarefas e é formado por dois componentes principais: uma interface homem-computador e uma aplicação (Figura 1.1). A *interface homem-computador* (de agora em diante denominada apenas interface) é a parte do sistema interativo responsável por: traduzir ações humanas em ativações de funções do sistema (aplicação), permitir que os resultados das ativações possam ser observados e coordenar esta interação. Em outras palavras, a interface é responsável tanto pelo mapeamento das ações do usuário (comandos e dados) para a aplicação como pela apresentação adequada dos resultados gerados. Já a *aplicação* é o elemento responsável pela parte funcional do sistema, que transforma dados de entrada em dados de saída através de computações, ou seja, é a parte responsável pelo processamento dos dados.

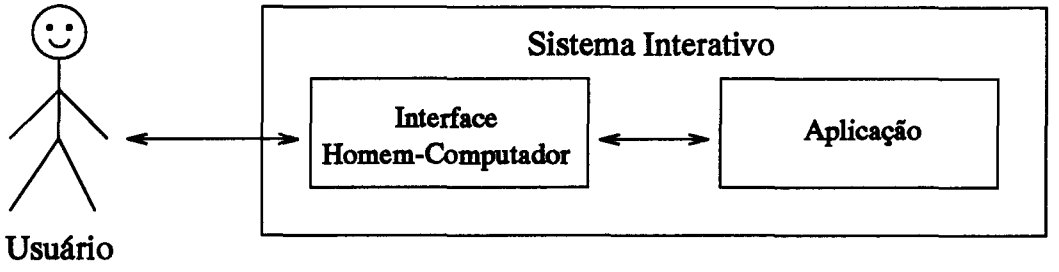



Figura 1.1: Visão simplificada de um sistema interativo

### 1.1.1 Independência de Diálogo

Na maioria das vezes, sistemas interativos são construídos combinando, sem nenhum critério, componentes da interface e da aplicação. Contudo, é cada vez maior o número de pessoas que concordam que deve haver algum tipo de separação entre esses componentes; essa separação é denominada *independência de diálogo* [31]. Para isso, há a necessidade da definição de mais um componente, denominado *diálogo*, na estrutura de sistemas interativos. Esse componente é responsável, através da definição de um protocolo de comunicação, pela coordenação das trocas de informações entre os outros dois componentes (*interface* e *aplicação*).

A independência de diálogo pode trazer vários benefícios para o desenvolvedor de sistemas interativos, dentre eles destacam-se os seguintes [11]:

- autonomia dos componentes; alterações em um componente tem influência limitada no outro;
- o desenvolvimento dos componentes pode ser realizado independentemente, exceto quanto à definição do protocolo de comunicação entre eles (diálogo);
- uma mesma aplicação pode ter interfaces diferentes desenvolvidas para grupos de usuários distintos. Língua, cultura, experiências e habilidades individuais de usuários ou até mesmo a execução em plataformas distintas podem motivar o desenvolvimento dessas diferentes interfaces.

Apesar das vantagens, a divisão de um sistema interativo em *interface*, *diálogo* e *aplicação* é reconhecidamente uma tarefa não trivial [32]. Além disso, tal separação pode gerar problemas de desempenho em sistemas que requerem intensa comunicação entre os dois componentes. Um exemplo são sistemas com manipulação de objetos em tempo real, onde um simples movimento de um dispositivo de interação (por exemplo, um *mouse*) pode gerar vários eventos em um curto espaço de tempo. Daí, surge a necessidade de uma arquitetura de software mais eficiente para a comunicação entre interface e aplicação, cujo

desempenho seja adequado. Caso contrário, o desempenho do sistema como um todo será afetado, podendo torná-lo inviável para certos tipos de aplicação.

### 1.1.2 Importância da Interface

Durante a escolha ou aceitação de um sistema interativo, a interface tem um papel importantíssimo. O usuário procura escolher o sistema que melhor atenda às suas necessidades, influenciado por aquele que o faça da maneira mais simples e agradável.

Além do fator de aceitação por parte do usuário, há também o fator custo. A eficiência do usuário no desenvolvimento das tarefas é fortemente influenciada pela interface do sistema. Uma interface de difícil assimilação dificulta a realização das tarefas. Além disso, a curva de aprendizado da interface torna-se mais acentuada e, conseqüentemente, um maior número de horas tem de ser despendido com treinamentos de usuários.

Outro fator que merece atenção é a redução da taxa de erros cometidos pelo usuário. Uma interface que conduz freqüentemente a erros pode diminuir de modo significativo o desempenho do usuário.

As considerações acima apontam para a necessidade de se projetar e construir interfaces denominadas amigáveis (*user-friendly*). Uma interface amigável possui, entre outras, as seguintes características [10]:

- **Facilidade de Uso e de Aprendizado** (*easy-to-learn and easy-to-use*). Aprender a usar uma interface geralmente implica em um investimento razoável de tempo. Uma boa interface reduz o custo de aprendizado e aumenta a produtividade do usuário. Interfaces deste tipo geralmente possuem algumas das seguintes características:
  - é transparente, ou seja, o usuário pode concentrar-se nas tarefas que necessita realizar, sem se preocupar com detalhes de funcionamento da interface;
  - é previsível;
  - é flexível;
  - e o usuário gosta dela.
- **Taxa de Erro Mínima.** É desejável minimizar a ocorrência de erros cometidos pelos usuários, mesmo porque erros afetam o desempenho de ambos usuários e sistema;
- **Recordação Rápida.** Idealmente, a seqüência de passos na execução das tarefas usuais deve ser entendida de forma natural pelo usuário. Normalmente, ele não deve ter que recorrer a manuais quando for usar o sistema;

- **Atrativa.** Nem sempre o sistema que possui maiores recursos a nível de aplicação é o preferido pelo usuário. Por exemplo, o usuário pode optar por um sistema cujo desempenho seja inferior, mas com o qual se sinta mais "confortável".

É desejável, portanto, que a interface de um sistema interativo tente integrar adequadamente todas essas características. Entretanto, os conflitos existentes entre tais características dificultam essa integração. Além disso, atualmente, a construção de interfaces ainda não possui um padrão de desenvolvimento sistemático que possa ser aplicado com sucesso garantido. Ou seja, não há uma regra comprovadamente eficaz que, se aplicada, possa assegurar a boa qualidade da interface gerada. Existe, também, a dificuldade de quantificação dessas características devido ao alto grau de subjetividade envolvido nas mesmas. Com o intuito de amenizar tais problemas, existem na literatura diretrizes[1] para construção de interfaces [52].

## 1.2 Modelagem Orientada a Objetos

A metodologia de projeto orientado a objetos adotada neste trabalho baseia-se na *Object Modelling Technique* (OMT) [50]. Dois dos três modelos propostos pela OMT são considerados essenciais em projeto orientado a objetos: o modelo de objetos e o modelo dinâmico, referenciados neste trabalho, respectivamente, como *modelo estrutural* e *modelo de controle*. Neste trabalho, um *modelo de objetos* é composto por um modelo estrutural e por um modelo de controle.

### 1.2.1 Fases da Metodologia Orientada a Objetos

Para o desenvolvimento de projetos orientados a objetos, a metodologia OMT apresenta quatro fases: análise, projeto orientado a objetos, projeto orientado a objetos e implementação.

#### Análise

A partir da delimitação de um problema do mundo real, constrói-se um modelo analítico que representa esse problema, de modo que se possa entendê-lo mais facilmente. Para isto, deve-se levantar os requisitos do problema, analisar as suas implicações e realizá-lo rigorosamente. Este modelo é apresentado em forma de objetos que abstraem as entidades do mundo real. Contudo, este modelo deve ser conciso e representar apenas "o que" o programa deve fazer e não "como" ele deve ser feito, isto é, não deve abordar nenhum aspecto de implementação, que é deixado para as fases seguintes.

[1] Em inglês: *guidelines*

### Projeto de Sistema

Baseado no modelo analítico, deve-se decidir, em alto nível, quais as características que devem ser otimizadas. Um projeto de sistema inclui decisões a respeito da organização do programa em subprogramas, alocação dos subprogramas aos componentes de *hardware* e *software* e estratégias de projeto. Tudo isto forma a base para o projeto detalhado. O resultado desta fase é um projeto que descreve as características e a funcionalidade do sistema de forma detalhada.

### Projeto Orientado a Objetos

Baseado no modelo analítico, que determina "o que" fazer, constrói-se um modelo de objetos contendo, agora, aspectos de implementação de acordo com a estratégia estabelecida durante a fase de projeto do sistema. O projeto orientado a objetos determina a definição completa das classes e relacionamentos entre as classes (modelo estrutural), bem como a definição dos estados e suas transições (modelo de controle).

### Implementação

O modelo de objetos (representando classes, relacionamentos, estados e transições) desenvolvido durante a fase de projeto orientado a objetos, é traduzido para programas na linguagem de programação alvo. Dentre as fases, esta deve ser relativamente menor e um processo basicamente "mecânico", pois todas as decisões que necessitam de uma análise mais apurada devem ser feitas durante as fases anteriores.

## 1.2.2 Conceitos e Notação Gráfica Associada

A seguir é apresentada a notação gráfica utilizada em cada um dos modelos, acompanhada de uma explicação sobre seu significado. Os campos que fazem parte da especificação gráfica do modelo estrutural e de controle podem ser obrigatórios ou opcionais. Campos obrigatórios aparecem entre parênteses angulares ("<>") e campos opcionais entre colchetes ("[ ]").

### Modelo Estrutural

O modelo estrutural [49] descreve o relacionamento entre as classes de objetos que compõem um programa orientado a objetos através de um diagrama (representação gráfica) contendo classes e seus respectivos atributos, métodos e relacionamentos. Este diagrama pode ser representado através de um grafo, cujo vértices representam classes e arestas representam relacionamentos entre classes.

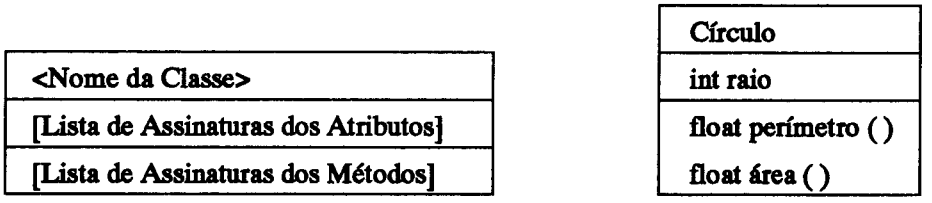



Figura 1.2: Notação de classe

### Classes

Uma classe descreve um grupo de objetos com propriedades similares (atributos), comportamento comum (métodos), semântica comum e relacionamento semelhante aos demais objetos. Uma classe é representada por um retângulo, subdivido em três sub-retângulos menores (Figura 1.2a). O sub-retângulo superior contém o nome da classe. O sub-retângulo central contém a lista de assinaturas dos atributos da classe e o inferior contém a lista de assinaturas dos métodos da classe. Tanto a lista de assinaturas de atributos quanto a lista de assinaturas de métodos são opcionais. A Figura 1.2b apresenta um exemplo de uma classe denominada Círculo, com atributo raio e métodos área e perímetro.

### Relacionamentos

No modelo estrutural podem existir quatro tipos de relacionamentos entre classes:

- **Generalização/Especialização**. O relacionamento de generalização/especialização é utilizado para expressar a herança entre uma classe base e uma classe derivada, onde a classe derivada herda atributos e comportamento (métodos) da classe base. De agora em diante o termo *herança* será utilizado como sinônimo de generalização/especialização. Este tipo de relacionamento é representado através de um triângulo que fica ligado à classe base e através de segmentos de reta que ligam o triângulo às suas classes derivadas (Figura 1.3a). Cada relacionamento de generalização/especialização pode conter opcionalmente, próximo ao triângulo, um nome que descreve o papel do relacionamento. Exemplo de relacionamento de generalização/especialização pode ser visto na Figura 1.3b, onde as classes derivada Elipse e Polígono, além das suas próprias especificações (atributos e métodos), herdam as especificações da sua classe base Figura Fechada, que por sua vez herda as especificações da sua classe base Figura.

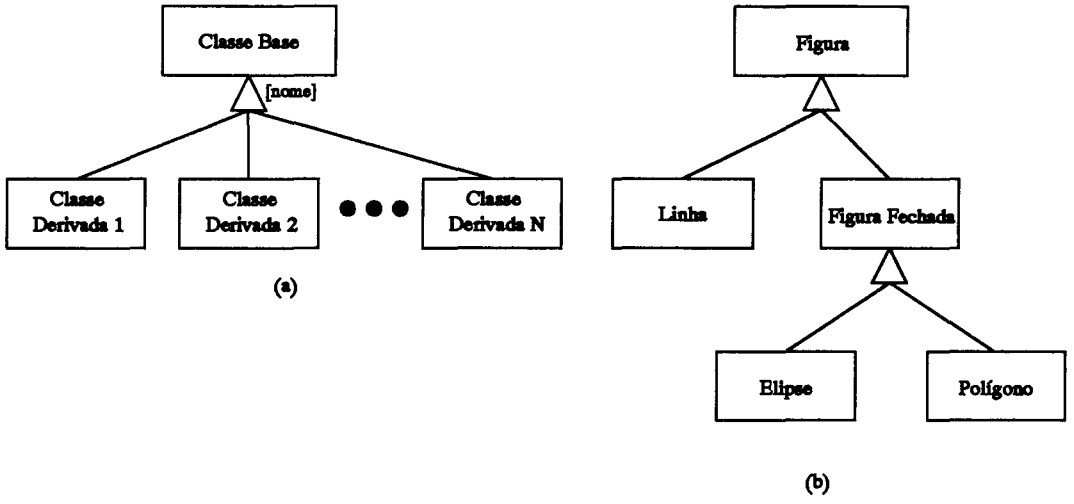



Figura 1.3: Notação do relacionamento de generalização/especialização no modelo estrutural

- **Associação.** O relacionamento de associação é utilizado para descrever que existe uma relação conceitual entre classes. Este relacionamento é representado através de segmentos de reta que ligam pares de classes (Figura 1.4a). Apenas relacionamentos binários são possíveis. Um relacionamento de associação pode possuir um nome, que é descrito próximo à mediana do segmento de reta que o representa. Cada classe pode possuir um papel no relacionamento, que pode ser descrito próximo à junção do segmento de reta com a classe. Além disso, cada classe possui uma cardinalidade, que especifica quantas instâncias da classe podem estar relacionadas a uma instância da outra classe associada. As cardinalidades também são representadas próximas à junção do segmento de reta com a classe, sendo abaixo do papel da relação, se este for descrito. Cardinalidades podem ser descritas através de um número fixo ("5"), faixa finita ("1-3") ou faixa infinita crescente ("1+"). No caso da omissão do valor da cardinalidade, é considerada cardinalidade um ("1"). Na Figura 1.4b, por exemplo, a classe Pessoa tem associação com a classe Empresa, indicando que uma pessoa pode estar associado a nenhuma ou várias empresas e que uma empresa pode estar associada a uma ou mais pessoas.

- **Agregação.** O relacionamento de agregação é utilizado para representar que uma classe tem um relacionamento do tipo "parte-todo" ou "uma-parte-de" com uma outra classe. As classes na relação "uma-parte-de" são denominadas classes componentes e a classe na relação "parte-todo" é chamada classe agregada, ou seja, uma classe maior (classe agregada) é composta por classes menores (classes componentes). Uma relação de agregação é representada através de um losango (Figura 1.5), que é representado ligado à classe agregada, e todas as classes derivadas são ligadas a

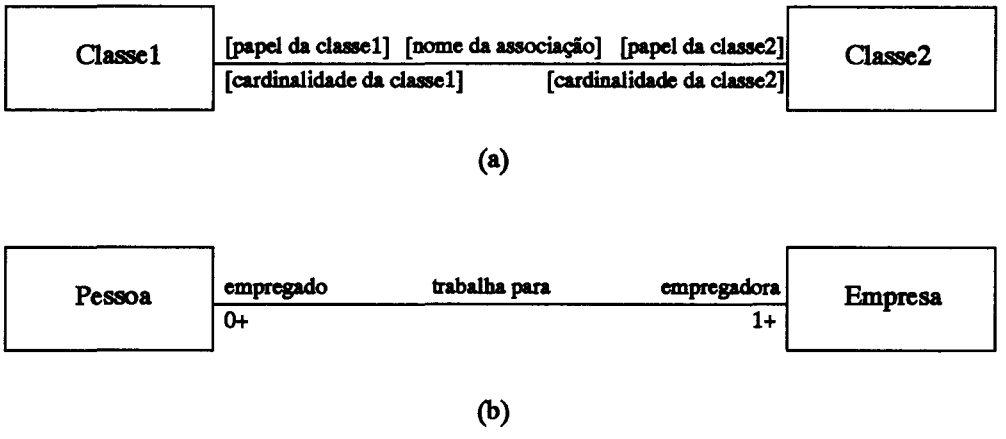



Figura 1.4: Notação do relacionamento de associação no modelo estrutural

este losango através de segmentos de reta. Semelhantemente à relação de associação, cada classe componente possui cardinalidade em relação a sua classe agregada, que pode ser descrito próximo ao ponto de junção do segmento de reta com a classe componente. Além disso, quando a existência de uma instância da classe componente depende da existência da instância da classe agregada, denomina-se de *agregação forte*, que é representada através de linhas sólidas (Figura 1.5a). Caso contrário, quando as instâncias das classes agregadas e componentes possuem existência independente, chama-se de *agregação fraca*, que é representada por segmentos de reta tracejados (Figura 1.5b). Na Figura 1.5c, por exemplo, a classe **Microcomputador** é composta pelas classes **Monitor**, **Teclado** e **Gabinete**. O que indica que microcomputadores são compostos de monitores, teclados e gabinetes. Além disso, indica que um microcomputador necessariamente só existe se esses três elementos estiverem agregados.

- **Metaclasse.** Este relacionamento é utilizado para representar classes que mantém informação sobre outra classe [7]. No modelo adotado, metaclasses apoiam a execução de mecanismos de reflexão computacional [38]. Uma arquitetura que implementa mecanismos de reflexão pode ser dividida em dois níveis: o nível básico, relacionado com a solução de problemas do domínio da aplicação e o nível reflexivo (metanível), relacionado com a solução de problemas e armazenamento de informações sobre o nível básico. Esta estrutura pode ser estendida recursivamente para conter mais níveis, ou seja, pode existir o meta-metanível, o meta-meta-metanível e assim por diante; cada metanível é associado a um metamodelo. Não é utilizada nenhuma notação gráfica para representar o relacionamento de metaclasse. Optou-se por representar o nível básico e os diferentes metaníveis em diferentes planos. Com isto, torna-se claro em que nível cada classe atua. Na Figura 1.6, a classe **Metaclasse**

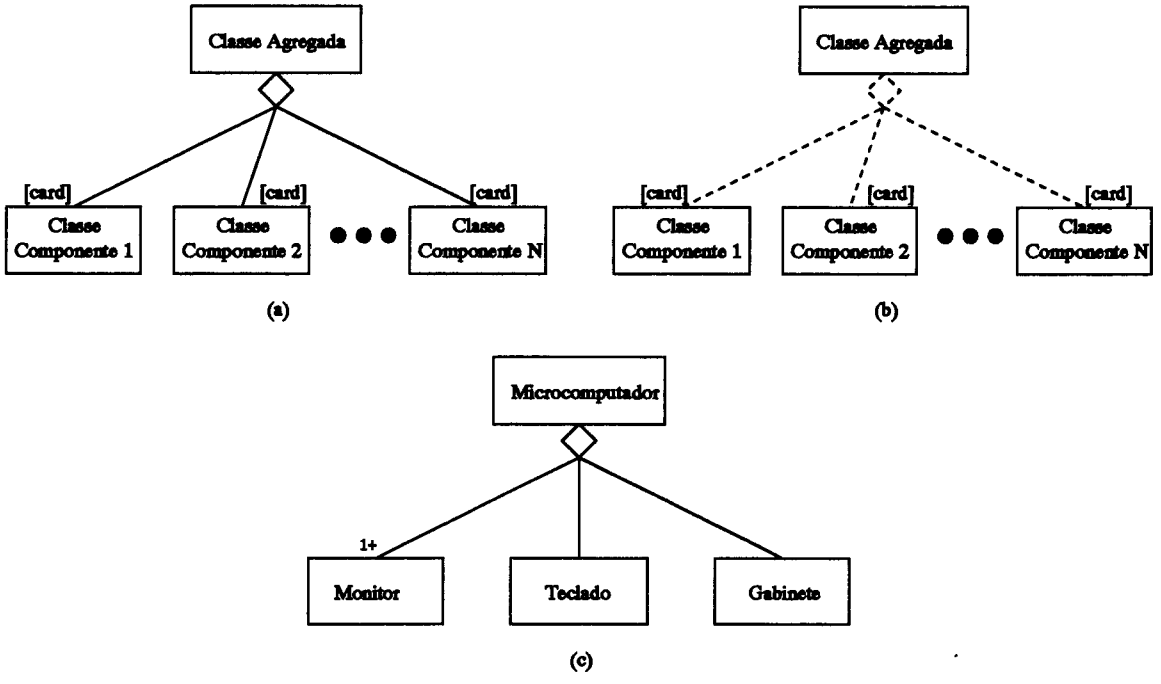



Figura 1.5: Notação do relacionamento de agregação no modelo estrutural

tem um relacionamento de metaclasse com a classe Classe Ordinária, representado simbolicamente, somente para efeito de visualização, através de uma linha tracejada orientada.

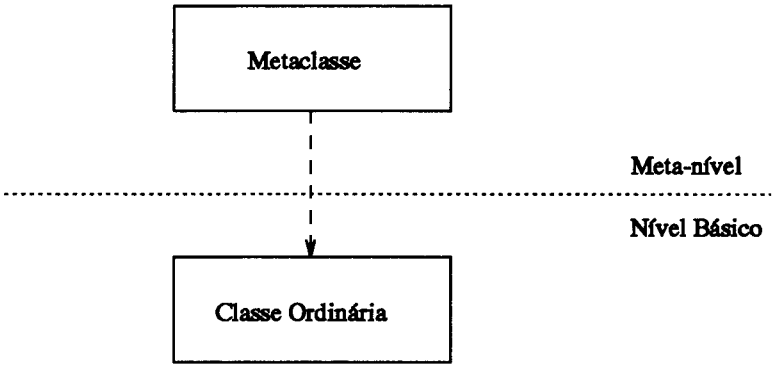



Figura 1.6: Exemplo de uma arquitetura reflexiva

### Modelo de Controle

O modelo de controle [48] reflete o relacionamento temporal entre os objetos de um programa distribuído e estão associados a classes do modelo estrutural. Modelos de controle

são descritos através de diagramas que representam máquinas de estados finitos, mais especificamente *statecharts* [29]. Estes diagramas podem ser representados como grafos, onde vértices do grafo representam estados e arestas representam transições entre os estados. Além de estados e transições, compõem o modelo de controle: eventos, condições, ações, encadeamentos de estados e estados *default*.

### Eventos

Um evento é um estímulo instantâneo (atômico) de um objeto a outro e que acontece em um determinado ponto do tempo, onde, estímulo é uma transmissão assíncrona de informação.

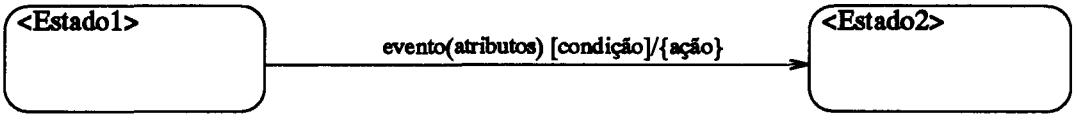



Figura 1.7: Notação geral do modelo de controle

### Estados

Um estado descreve a configuração dos atributos de um objeto durante um intervalo de tempo. Cada estado é representado através de um retângulo com bordas arredondadas e possui necessariamente um nome associado a ele, que é representado no canto superior esquerdo do retângulo (Figura 1.7). Este nome deve ser único em cada modelo de objetos.

### Transições

Uma transição é uma resposta a um evento, que pode invocar uma ação e pode causar uma mudança de estado no objeto. Uma transição é representada por uma linha orientada (Figura 1.7).

### Condições

Condição é uma expressão booleana, que pode ser usada como guardiã de transições, ou seja, quando um evento ocorre, a transição a ele associada somente é disparada se a condição de guarda for verdadeira. Condições são representadas entre colchetes (Figura 1.7).

### Ações

Ação é uma operação instantânea (atômica). Uma ação é associada a uma transição e é executada toda vez que a transição a ela associada for disparada. Sua representação é feita entre chaves após uma barra (Figura 1.7).

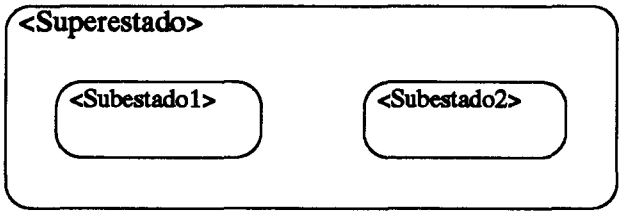



Figura 1.8: Notação do relacionamento de generalização/especialização no modelo de controle

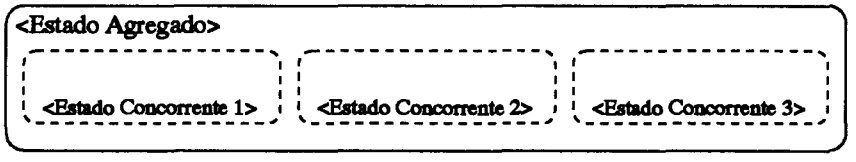



Figura 1.9: Notação do relacionamento de agregação no modelo de controle

### Estados Encadeados

Diagramas de estados podem ser estruturados de forma a permitir uma descrição estruturada de sistemas complexos. A maneira de estruturar máquinas de estados é similar à maneira utilizada para estruturar hierarquia de classes. Hierarquias de estados podem ser formadas utilizando-se dois tipos de estados encadeados: "ou-exclusivos" ou "concorrentes". No modelo estrutural, o relacionamento de generalização/especialização é utilizado para modelar classes bases e classes derivadas. Similarmente, no modelo de controle, generalização/especialização permite que estados e eventos sejam arranjados em hierarquias com herança de estrutura e comportamento comuns. Este tipo de relação é do tipo "ou-exclusivo". Por exemplo, na Figura 1.8, estar no estado **Superestado** indica estar necessariamente em um dos estados **Subestado1** ou **Subestado2**, mas não em ambos. Já a agregação permite que um estado seja dividido em componentes independentes, como acontece no modelo estrutural. Estes componentes independentes possuem execução concorrente e por isso são denominados estados "concorrentes" (Figura 1.9).

O relacionamento de generalização/especialização é representado através da disposição dos subestados dentro do superestado, ou seja, o superestado inclui geometricamente seus

subestados (Figura 1.8). A relação de agregação também é representada pela inclusão geométrica dos subestados pelo superestado, indicando que o superestado está divido em subestados concorrentes. Contudo, os subestados são representados na forma tracejada (Figura 1.9), ao contrário da relação de generalização/especialização, onde os subestados são representados na forma contínua (Figura 1.8).

Um estado não pode ter simultaneamente como subestados imediatos estados concorrentes e exclusivos.

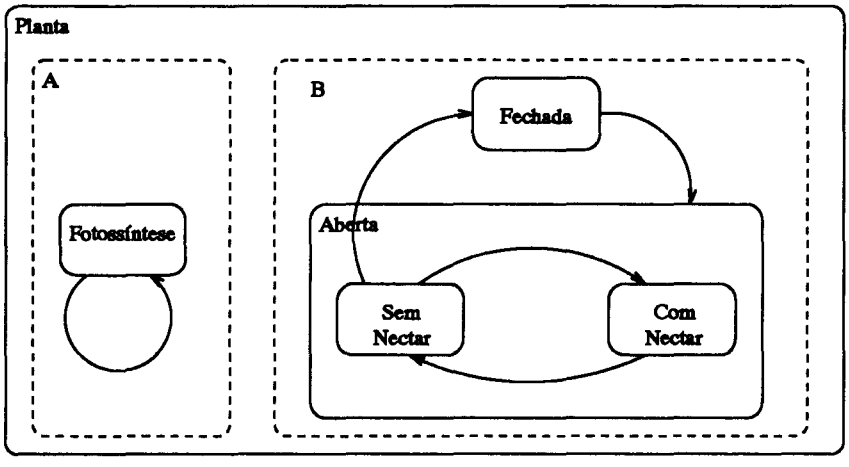



Figura 1.10: Exemplo de um diagrama do modelo de controle

Por exemplo, na Figura 1.10, os estados Sem Néctar e Com Néctar estão incluídos no estado Aberta e por isso os estados Sem Néctar e Com Néctar herdam as especificações do estado Aberta, que por sua vez, juntamente com o estado Fechada, herda as especificações do estado Planta. O estado Planta é subdividido nos estados concorrentes A e B, portanto a execução do estado planta pode ser feito através da execução concorrente de seus subestados A e B.

### Estados *Default*

Em uma hierarquia de estados exclusivos, quando o estado pai torna-se ativo, automaticamente um de seus subestados deve tornar-se ativo também. Para a resolução de qual de seus subestados deve tornar-se ativo, é necessária a indicação de um estado denominado estado default. Sua indicação é feita através de um círculo preenchido ligado a ele através de um segmento de reta orientado. Na Figura 1.11, por exemplo, se o estado A tornar-se ativo, automaticamente o estado B torna-se ativo.

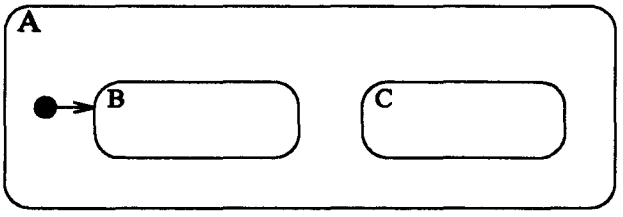



Figura 1.11: Exemplo de definição de estado *default*

## 1.3 Leiaute Automático

É cada vez maior a utilização de ferramentas com a finalidade de auxiliar na construção de leiautes de diagramas em uma determinada mídia (por exemplo, tela de computador, impressora, etc). Na ausência de tais ferramentas, a construção do leiaute tem de ser realizada manualmente. Contudo, tal tarefa é morosa, tediosa e algumas vezes difícil de ser realizada, principalmente quando envolve um número razoável de elementos gráficos. Por este motivo, a construção de tais tipos de ferramentas pode ser considerada de grande utilidade.

Exemplos de uso desse tipo de ferramenta podem ser encontrados, principalmente, em diversas áreas da ciência da computação, tais como: projeto de circuitos integrados, engenharia de *software* (diagramas de entidade-relacionamentos [59], de fluxo de dados [1, 43]), redes Pert, representação de conhecimento, entre outras.

Há na literatura uma diversidade enorme de algoritmos para leiaute de diagramas. Devido a esta variedade, há a necessidade de critérios para avaliar a qualidade dos leiautes gerados por esses algoritmos, na tentativa de classificá-los. Inicialmente, eles podem ser classificados em duas categorias [53]:

- Enfatizando aspectos de implementação física, tais como: confiabilidade e economia de recursos. Tais algoritmos são freqüentemente utilizados em leiaute de circuitos integrados;
- Enfatizando aspectos da visão cognitiva humana, tais como: legibilidade e estética.

Dado que neste trabalho pretende-se abordar apenas aspectos como legibilidade e estética e que, além disso, os diagramas a serem traçados podem ser representados como grafos, restringe-se o enfoque apenas a algoritmos de leiaute de grafos que enfatizam legibilidade e estética.

Nas subseções seguintes, apresentam-se os conceitos básicos de grafos [57, 2] utilizados durante a apresentação deste trabalho, bem como uma rápida apresentação da área e dos problemas relacionados com leiaute de grafos.

### 1.3.1 Grafos

Um *grafo* $G = (V, E)$ é um conjunto finito não vazio $V$ de *vértices* e um conjunto $E$ de *arestas*, que são pares não ordenados de elementos distintos de $V$, ou seja, $E$ define uma relação binária sobre $V$. Cada aresta $e \in E$ é denotada pelo par de vértices $e = vw$. Um grafo $G' = (V', E')$ é denominado *subgrafo* de $G = (E, V)$ se $V' \subseteq V$ e $E' \subseteq E$.

Em alguns casos exige-se que arestas em um grafo G tenham seus extremos distintos e que duas arestas não possuam o mesmo par de vértices, neste caso diz-se que G é um grafo simples.

Dados dois vértices $v$ e $w$ em um grafo G, denomina-se *caminho* uma seqüência $S = \{v = u_1, u_1u_2, u_2, u_2u_3, \ldots, u_{k-1}u_k, u_k = w\}$ que não repita vértices. O *tamanho de um caminho* (ou comprimento) é o número de arestas de S e é denotado por $Cam(v, w)$. Um *ciclo* é um "caminho fechado", isto é, permite-se que somente o último vértice do caminho seja repetido.

Um grafo é denominado *conexo* quando existe caminho entre cada par de vértices do grafo. Caso contrário é denominado *desconexo.*

Um dígrafo $D = (V, E)$ é um grafo onde as arestas $e \in E$ são pares ordenados $(v, w)$. Num dígrafo, cada aresta $(v, w)$ possui uma única direção de $v$ para $w$, onde $w$ é a *cabeça* da aresta e $u$ a *cauda.* Diz-se também que a aresta $(v, w)$ é *divergente* de $v$ e *convergente* em $w$. De agora em diante, utiliza-se arestas como sinônimo de arestas orientadas.

Se para $v, w \in V$ existe um caminho de $v$ para $w$, ou seja, uma seqüência $v = u_1, (u_1, u_2), u_2, (u_2, u_3), \ldots, (u_{k-1}, u_k), u_k = w$ então diz-se que $v$ *atinge* $w$ e que $w$ é *atingível* por $v$.

Um dígrafo $D = (V, E)$ é *fortemente conexo* quando para todo par de vértices $v, w \in V$ existir um caminho de $v$ para $w$. Um *componente fortemente conexo* de um dígrafo $D = (V, E)$ é um subgrafo $D' = (V', E')$ onde $D'$ é um dígrafo fortemente conexo.

O *grau de entrada* de um vértice $v$ é o número de arestas convergentes em $v$. O *grau de saída* de $v$ é o número de arestas divergentes de $v$. A soma do grau de entrada e do grau de saída é denominado *grau.* Uma *fonte* é um vértice com grau de entrada igual a zero.

Dado um dígrafo $D = (V, A)$, diz-se que um grafo $G = (V, E)$ é um *grafo subjacente* ao dígrafo $D$, se $E$ é obtido pela remoção das orientações das arestas de $A$.

Um dígrafo $D$ é dito ser uma *árvore enraizada* se o seu grafo subjacente é acíclico e conexo e existe um vértice denominado *raiz* que atinge todos os vértices em $D$. Doravante, diz-se apenas árvore referenciando árvore enraizada. Os vértices que possuem grau de saída igual a zero são denominados *folhas.* A profundidade de um vértice $v$ na árvore é o tamanho do caminho do vértice raiz até o vértice $v$ e é denotado por $Prof(v)$.

Em uma árvore, *pai*, *ancestrais*, *filhos* e *descendentes* de um vértice $v \in V$ são denotados respectivamente por: $Pai(v)$, $Ancestrais(v)$, $Filhos(v)$ e $Descendentes(v)$. E são

definidos da seguinte maneira:

$$Pai(v) = \{u \in V | u \text{ atinge } v \text{ e } Cam(u,v) = 1\}$$
$$Ancestrais(v) = \{u \in V | u \text{ atinge } v\} \cup \{v\}$$
$$Filhos(v) = \{u \in V | v \text{ atinge } u \text{ e } Cam(v,u) = 1\}$$
$$Descendentes(v) = \{u \in V | v \text{ atinge } u\} \cup \{v\}$$

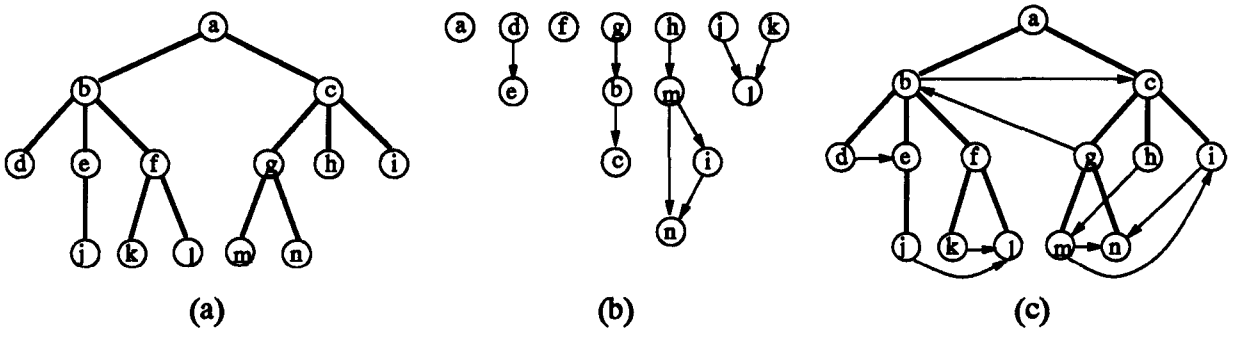



Figura 1.12: Exemplo de um dígrafo composto

Dado dois tipos de relações binárias, relações de inclusão e adjacência, definidas sobre um conjunto de vértices $V$, pode-se definir dois tipos de dígrafos, correspondentes a essas relações. Um *dígrafo de inclusão* $D_I = (V, E)$, onde $(u, v) \in E$ significa que $u$ inclui $v$ e é denominada *aresta de inclusão* (Figura 1.12a). E um *dígrafo de adjacência* $D_A = (V, F)$, onde $(u, v) \in F$ significa que $u$ é adjacente a $v$ e é denominada *aresta de adjacência* (Figura 1.12b). Um *dígrafo composto* $C = (V, E, F)$ é obtido pela composição destes dois dígrafos (Figura 1.12c). Como se pode notar, a Figura 1.12 omite as orientações do dígrafo de inclusão, que devem ser subentendidas como sendo de cima para baixo. A supressão da orientação facilita a leitura do dígrafo. De agora em diante, esta convenção é utilizada para todo o dígrafo composto.

### 1.3.2 Visão Geral da Área de Desenho de Grafos

Desenho de Grafos[2] é a área que aborda o problema de construir leiautes de grafos com alto grau de legibilidade, ou seja, com "bons" leiautes. Informalmente, um leiaute é considerado "bom" se ele é capaz de transmitir de forma clara e precisa a informação por ele representado de forma gráfica. Uma melhor visão sobre a área de Desenho de Grafos pode ser encontrada em [58].

Dado o alto grau de subjetividade envolvido na avaliação de quão "bom" um leiaute possa ser, surge a necessidade da definição de critérios, na tentativa de se expressar algum

[2]Em inglês: *Graph Drawing*

tipo de métrica que possa ajudar durante a quantificação desses leiautes. Por exemplo, na Figura 1.13 são representados três possíveis leiautes para um mesmo dígrafo. Se um grupo de pessoas pudesse escolher, dentre os três, o leiaute que mais lhe agrade, provavelmente a escolha não seria unânime. Algumas poderiam escolher o leiaute da Figura 1.13c por se parecer ao formato de um dado. Outras, no entanto, talvez poderiam escolher o leiaute da Figura 1.13a, pelo fato de não haver nenhum cruzamento entre linhas ou a Figura 1.13b, por ela apresentar um alto grau de simetria.

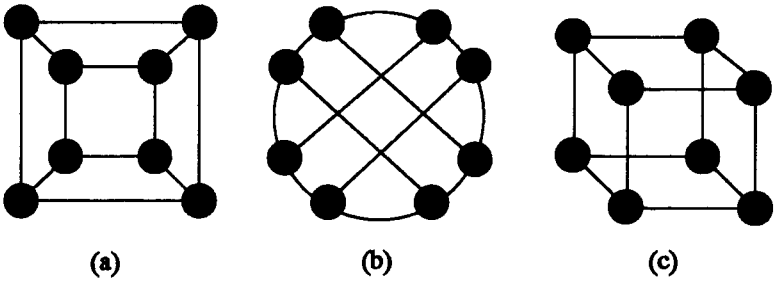



Figura 1.13: Exemplo da existência de subjetividade na avaliação de leiautes

Dado a dificuldade de medir de forma exata o quão "bom" um leiaute possa ser, dois critérios são considerados importantes na tentativa de expressar esta medida: características estéticas e restrições semânticas.

*Características estéticas* são utilizadas para denotar critérios relativos a legibilidade. Tais características são independentes da informação a ser representada. A seguir são apresentadas algumas das características estéticas que podem ser levadas em consideração durante a avaliação de um leiaute [59]:

E1. minimização da área ocupada;

E2. balanceamento do leiaute em relação ao eixo horizontal ou eixo vertical;

E3. minimização do número de "quebras" (*bends*) ao longo de uma aresta;

E4. maximização do número de faces desenhadas como polígonos convexos;

E5. disposição no centro do leiaute dos vértices com alto grau;

E6. minimização das diferenças entre as dimensões dos vértices;

E7. minimização do tamanho total das arestas;

E8. uniformização do tamanho das arestas;

E9. minimização do tamanho da aresta mais longa;

E10. maximização da simetria do leiaute;

E11. simetria dos vértices filhos em hierarquias;

E12. distribuição uniforme dos vértices no leiaute;

E13. minimização de cruzamentos entre arestas;

E14. verticalidade das estruturas hierárquicas;

E15. balanceamento uniforme do grau angular entre as arestas convergentes (divergentes) de um vértice;

E16. disposição dos vértices ligados entre si o mais perto possível uns dos outros;

E17. minimização de cruzamentos entre arestas e vértices.

É importante ressaltar que um leiaute "bom" em relação a uma característica estética pode ser "ruim" em relação a outra, ou seja, tais características algumas vezes são conflitantes. Por exemplo, na Figura 1.14 são apresentados dois possíveis leiautes para um mesmo dígrafo. O leiaute apresentado na Figura 1.14a minimiza o número de cruzamento de arestas. Já o leiaute da Figura 1.14b maximiza a simetria. Neste caso, é impossível otimizar simultaneamente ambas as características estéticas. Pode-se encontrar na literatura, estudos realizados de modo a descobrir a eficiência da aplicação destas características estéticas na legibilidade de leiautes de grafos [44]. Idealmente, um algoritmo deve levar em conta cada um das características que procura atender, de acordo com o problema a ser atacado.

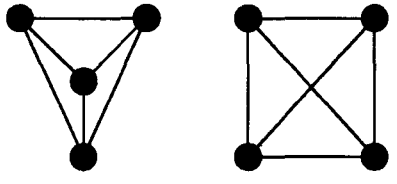



Figura 1.14: Exemplo da impossibilidade de otimizar simultaneamente certas características estéticas

*Restrições semânticas* geralmente são utilizadas para expressar características que são dependentes de contexto. A seguir são apresentadas algumas restrições encontradas em algoritmos de leiaute de grafos [59]:

R1. posicionar um dado conjunto de vértices no centro do leiaute;

R2. associar dimensões aos símbolos que representam os vértices;

R3. posicionar determinados vértices no limite externo do leiaute;

R4. posicionar próximos um determinado grupo de vértices;

R5. desenhar um subgrafo com uma forma específica;

R6. restringir o leiaute a uma determinada área;

R7. posicionar uma seqüência de vértices ao longo de uma linha reta.

Durante a representação de um leiaute, vértices são na maioria das vezes representados por retângulos, pontos, círculos ou elipses. Arestas são representadas por curvas contínuas ligando um vértice a outro; setas são utilizadas para representar a sua orientação. Vários padrões de desenho têm sido propostos na literatura [16] de modo a classificar os diferentes tipos de leiaute de grafos. A seguir apresenta-se alguns tipos comumente encontrados em Desenho de Grafos:

- **Reta** (*Straight-line*). Cada aresta é representada por um único segmento de reta (Figura 1.15b);
- **Segmentos de Reta** (*Polyline*). Cada aresta é representada por uma série encadeada de segmentos de retas (Figura 1.15a);
- **Ortogonal**. Cada aresta é representada por uma série de segmentos horizontais e verticais de reta (Figura 1.15c);
- **Grid**. É uma representação *polyline* onde vértices, cruzamentos entre arestas e quebras têm coordenadas inteiras;
- **Planar**. Não existe cruzamento entre arestas (Figura 1.14a);
- **Upward**. Cada aresta é representada na forma não decrescente em relação a uma dada direção (Figura 1.15d);
- **Hierarquia**. Vértices são dispostos em camadas onde toda aresta liga vértice de uma cada superior a uma camada inferior. Todos vértices dispostos em uma mesma camada são representados na mesma posição horizontal (Figura 1.12a).

Alguns algoritmos tentam resolver o problema de geração de leiaute para qualquer classe de grafos (grafos gerais). Contudo, tais algoritmos, na maioria das vezes, são muito lentos dada a complexidade do problema envolvido. Além disso, certas características

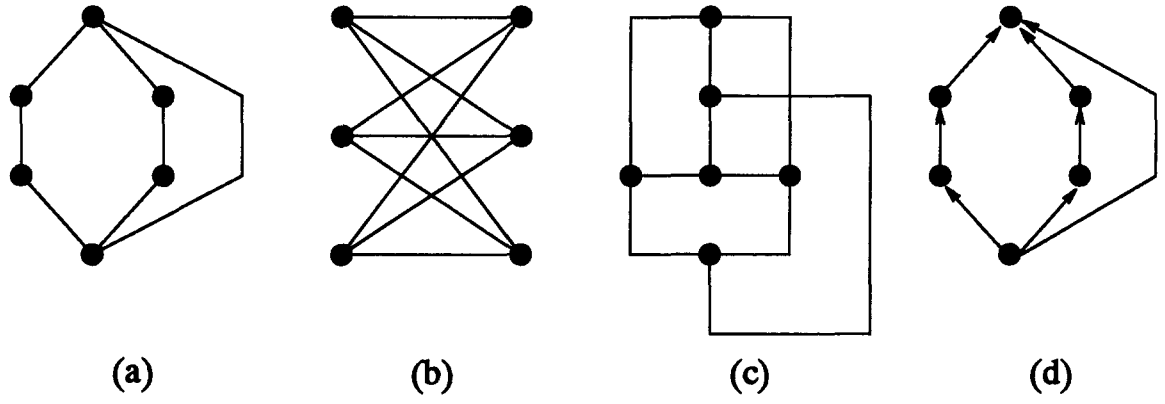



Figura 1.15: Padrões de desenho de grafos

estéticas dependem do padrão de desenho adotado e da particular classe de grafo interessada. Assim, uma parte dos algoritmos restringem sua atuação a uma determinada classe de grafos. Alguns destes algoritmos são restritos a dígrafos, pois as orientações das arestas influenciam no leiaute gerado.

As atuais técnicas para geração de leiaute de grafos são baseadas basicamente em dois tipos de abordagens: a abordagem algorítmica e a abordagem declarativa.

A *abordagem declarativa* permite especificar um conjunto de restrições de modo que o leiaute gerado satisfaça, ou melhor dizendo, tente satisfazer tal conjunto. Este tipo de abordagem tem a vantagem de ser flexível na especificação do tipo de leiaute desejado. Sua utilização permite adicionar ou remover certas restrições na tentativa de melhorar a solução a ser gerada. Contudo, geralmente, as técnicas baseadas nesta abordagem são computacionalmente mais ineficientes, devido à complexidade inerente envolvida na resolução de restrições. Além disso, certas características estéticas necessitam de complicadas restrições para serem devidamente expressas.

A *abordagem algorítmica* enfatiza o desenvolvimento de algoritmos para geração de leiautes que satisfaçam apenas um número fixo e pré-determinado de características estéticas. Sua vantagem está na eficiência computacional.

Contudo, nem sempre a aplicação de uma única abordagem isoladamente produz o resultado esperado. Por isto, a utilização de mais de um tipo de algoritmo pertencente a diferentes abordagens pode gerar resultados mais satisfatórios. Exemplos deste tipo de técnica podem ser encontrados em [15, 18]. Além disso, há tentativas de integração das duas abordagens, de modo que as vantagens de uma possam compensar as desvantagens da outra [19]. Exemplos de técnicas que tentam conciliar as duas abordagens podem ser encontrados em [30].

Na Tabela 1.1 são apresentados alguns dos algoritmos de leiaute de grafos encontrados na literatura. Um estudo mais detalhado desses algoritmos pode ser encontrado em [16].

| Referência | Classe de Grafo | Padrão de Desenho | Característica Estética | Restrição Semântica |
|---|---|---|---|---|
| Sugiyama e Misue [53] | dígrafos compostos | *polyline*, *upward* e hierarquia | E3, E13, E15, E16 e E17 | |
| Sugiyama, Tagawa e Toda [56] | dígrafos gerais | *polyline*, *upward* e hierarquia | E3, E13, E15 e E16 | |
| Davidson e Harel [9] | grafos gerais | reta | E8, E12 e E13 | R6 |
| Fruchterman e Reingold [24] | grafos gerais | reta | E8, E10, E12 e E13 | R6 |
| Tamassia, di Battista e Batini [59] | grafos gerais | *grid* e hierarquia | E1, E3, E7 e E10 | R1, R3, R4, R5 e R7 |
| Gschwind e Murtagh [28] | dígrafos gerais | hierarquia, *upward* e *polylines* | E3, E13 e E16 | |
| Tunkelang [61] | grafos gerais | reta | E8, E12 e E13 | |
| Kamada e Kawai [34] | dígrafos gerais | reta | E10, E12 e E13 | |
| Chiba, Onoguchi e Nishizeki [6] | grafos planares | reta | E4 | |
| Batini, Nardelli e Tamassia [1] | grafos gerais | *grid* | E1, E3, E7 e E13 | R3 |
| Gansner, Koutsofios, North e Vo [25] | dígrafos gerais | *polyline*, *upward* e hierarquia | E3, E7, E10, E13 e E15 | |

Tabela 1.1: Alguns algoritmos para leiaute de grafos

## 1.4 Objetos e Ações

Mecanismos de sincronização como semáforos [60] são essencialmente mecanismos de baixo nível que requerem que o programador esteja intimamente envolvido com todos os detalhes de exclusão mútua, região crítica e *deadlocks*.

Por este motivo, outros mecanismos foram desenvolvidos no sentido de tornar disponível uma técnica de sincronização com um nível um pouco mais alto de abstração. Um exemplo de tal mecanismo é transação atômica[3]. Ação atômica, ou simplesmente ação, é utilizado como sinônimo de transação atômica.

Este mecanismo surgiu através de pesquisas na área de gerência de bancos de dados. Porém, mais tarde surgiram os primeiros trabalhos que estenderam o uso de ações atômicas para a programação de sistemas distribuídos. O objetivo é ajudar a diminuir a complexidade na construção de programas em ambientes distribuídos.

Ações atômicas possuem três propriedades fundamentais [60]:

- **Equivalência Serial.** Esta propriedade garante que, se duas ou mais ações estão executando concorrentemente sobre objetos compartilhados, o resultado final é equivalente a uma execução das ações em alguma ordem serial;
- **Atomicidade.** Esta propriedade garante que uma ação, ou é consolidada[4] terminando normalmente e gerando o resultado desejado, ou então é abortada[5] e não produz resultado;
- **Permanência.** Esta propriedade garante que o resultado gerado por uma ação consolidada torna-se permanente e nenhuma falha que possa vir a acontecer posteriormente pode desfazê-lo ou causar a sua perda.

A propriedade de ação atômica é utilizada para controlar as mudanças de estados de objetos persistentes e garantem que somente transformações consistentes de estados aconteçam, mesmo na presença de falhas.

Em ambientes de programação transacional e orientada a objetos, programas distribuídos são considerados coleções de objetos cujos métodos são atômicos, isto é, todo método de um objeto é implementado como uma ação atômica. Muitos ambientes dessa classe provêem ações atômicas encadeadas na sua interface de programação. Porém, a propriedade de permanência somente se aplica à ação de nível mais alto.

[3]Do inglês: *atomic transaction*
[4]Do inglês: *commited*
[5]Do inglês: *aborted*

## 1.5 Resumo

Neste capítulo foram descritos, de maneira sucinta, os conceitos que, de alguma forma, fundamentaram o desenvolvimento deste trabalho. Inicialmente, foram apresentados conceitos relativos a sistemas interativos, onde foram comentados alguns aspectos da importância de interfaces e das vantagens da obtenção de independência de diálogo neste tipo de sistema.

Em seguida, foram apresentados conceitos da metodologia orientada a objetos os quais este trabalho foi baseado. As fases normalmente necessárias para o desenvolvimentos de sistemas utilizando-se essa metodologia, bem como toda a sua notação gráfica também foram abordadas.

Posteriormente, uma rápida visão da área de Desenho de Grafos foi apresentada, com ênfase principalmente em seus objetivos. Além disso, com o intuito de mostrar a diversidade de trabalhos existentes nessa área, foi apresentada uma pequena relação de algoritmos para leiaute de grafos. Para fundamentar as discussões relativas a área de Desenho de Grafos foram apresentados alguns conceitos relativos a grafos.

Finalmente, apresentou-se conceitos relativos a objetos e ações, com ênfase principalmente na definição de ação atômica e como estes conceitos se inter-relacionam dentro do escopo deste trabalho.

# Capítulo 2

# Editor Gráfico

O editor gráfico descrito neste trabalho foi concebido visando dar subsídios ao ambiente de programação distribuída Stabilis/Vigil, que, por sua vez, foi implementado sobre um sistema de suporte a programação distribuída e tolerantes a falhas, denominado Arjuna. Por se tratar de um sistema interativo, a estrutura de *software* do editor gráfico foi projetada em vistas a obter as vantagens fornecidas pelo conceito de independência de diálogo.

O objetivo deste Capítulo é apresentar as arquiteturas de cada um dos sistemas que compõem o ambiente de programação distribuída, além do processo envolvido em torno do desenvolvimento de uma aplicação para esse ambiente. Em seguida, é apresentada em linhas gerais a arquitetura do editor gráfico, bem como a utilização do editor no processo de construção de aplicações para o ambiente de programação distribuída. Contudo, todo o detalhamento da estrutura interna do editor é deixado para os capítulos 3 e 4.

## 2.1 Arquitetura de Software do Ambiente

O ambiente é composto pelo sistema Stabilis/Vigil em conjunto com o sistema Arjuna. Estes sistemas formam a base para o ambiente de programação distribuída, tolerante a falha e orientada a objetos, o qual o editor visa dar apoio. A seguir, descreve-se a arquitetura de ambos os sistemas, bem como a do editor gráfico.

### 2.1.1 Arjuna

Arjuna [14, 42] é um sistema de programação orientada a objetos totalmente implementado em C++, que provê um conjunto de ferramentas para a construção de aplicações distribuídas e tolerantes a falhas. Todo o sistema Arjuna é baseado no modelo de objetos e ações. Além disso, ele implementa ações atômicas encadeadas, controle de concorrência, recuperação de estado e persistência. Todos estes mecanismos formam a base para tornar

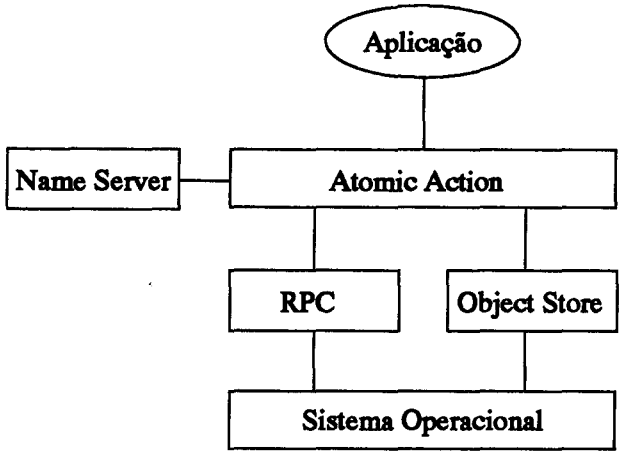



Figura 2.1: Arquitetura da plataforma distribuída Arjuna

uma aplicação tolerante a falhas. Arjuna os torna disponíveis ao programador através de suas classes. Tais conceitos podem ser embutidos de modo seletivo, de acordo com a necessidade da aplicação, através do mecanismo de herança.

Toda operação em Arjuna pode ser controlada por ações atômicas. Para o tratamento de tolerância a falhas, Arjuna supõe que os componentes de *hardware* do sistema são estações de trabalho conectadas por um subsistema de comunicação e que estas estações oferecem uma semântica de falha do tipo "falha e parada"[1].

Arjuna foi programado em C++, que é a linguagem para a qual o editor gráfico traduz os modelos de objetos capturados. Sua interface de programação também é baseada na linguagem C++.

A arquitetura de Arjuna é composta por vários módulos (Figura 2.1), que são descritos a seguir:

- **Módulo Object Store.** Fornece acesso a um serviço de gerência de estados persistentes de objetos. A representação estável de um objeto persistente, normalmente armazenada em disco, deve ser independente de arquitetura de *hardware* para permitir a sua transmissão como uma mensagem. A classe ObjectState implementa tal representação, provendo operações para a manipulação do estado persistente de um objeto. A função do módulo Object Store é manipular e administrar instâncias da classe ObjectState;

- **Módulo RPC.** Provê facilidades para a invocação de operações sobre objetos persistentes, dentro do modelo cliente-servidor. Um servidor gerencia o estado dos objetos; ele define e executa operações que são exportadas para os clientes. Os

[1] Do inglês: *failstop*

clientes ativam essas operações a fim de alterar o estado do objeto gerenciado pelo servidor. Todas as invocações de operações são implementadas como chamadas de procedimento remoto (*remote procedure call*) cuja execução está a cargo do módulo RPC;

- **Módulo Name Server.** Mantém informação sobre a identificação dos objetos (nomes) e a sua localização. Nomes são utilizados para fazer referência a objetos. A função de gerência mapeia nomes legíveis dado por usuários para identificadores únicos de objetos. A função de resolução mapeia o identificador único do objeto para a sua localização no sistema distribuído;

- **Módulo Atomic Action.** O módulo Atomic Action fornece a interface de programação do sistema Arjuna. Os mecanismos necessários para controle de concorrência e controle de ações atômicas são implementados pelas classes contidas neste módulo (Figura 2.2). Estas classes representam parte do modelo estrutural que compõe a estrutura interna do módulo.

  As operações **begin**, **end** e **abort**, fornecidas pela classe AtomicAction são utilizadas para implementar métodos atômicos. A existência de ações atômicas encadeadas permite um controle flexível da atomicidade. Contudo, os únicos objetos controlados pelas ações atômicas são os objetos pertencentes ao sistema Arjuna ou os objetos cujas classes são derivadas da classe LockManager (Figura 2.2).

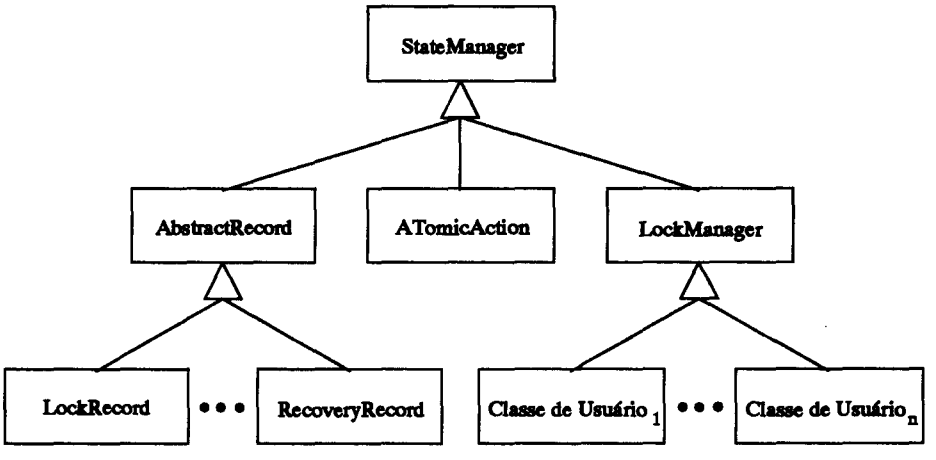



Figura 2.2: Parte modelo estrutural do módulo Atomic Action

## 2.1.2 Stabilis/Vigil

O sistema Arjuna, através do seu modelo computacional baseado em ações atômicas, fornece um ambiente propício para implementação de programas distribuídos e tolerantes

a falhas. Apesar disto, há a necessidade de ferramentas que auxiliem o programador não só durante a fase de implementação, mas também durante as demais fases de construção de um programa, como por exemplo, gerência e projeto.

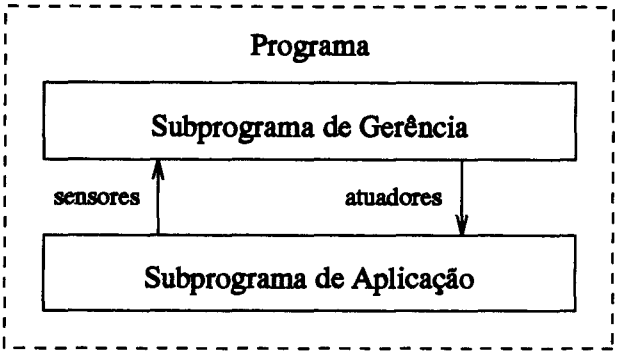



Figura 2.3: Programa visto como um sistema reativo

No ambiente de programação descrito neste trabalho, um programa é visto como um sistema reativo, que é composto por dois subprogramas que interagem: Aplicação e Gerente (Figura 2.3). A interação entre esses subprogramas é realizada por intermédio de atuadores e sensores. *Sensores* são utilizados para obter e avaliar o estado de um objeto em um dado momento. *Atuadores* são métodos utilizados para modificar o estado de um objeto. As políticas de um programa são descritas no subprograma Gerente. Esse subprograma, através de atuadores e sensores, pode então controlar e alterar o comportamento do subprograma Aplicação de acordo com as políticas adotadas.

A abstração de programa distribuído apresentada anteriormente contribui fortemente para a criação de um ambiente de programação composto por dois subsistemas: Stabilis e Vigil [4]. Neste ambiente, os conceitos de objetos, classes, relacionamentos, estados e transições são utilizados para formar um modelo de objetos de um programa distribuído. Ambos os subsistemas foram implementados utilizando as ferramentas fornecidas pelo sistema Arjuna. A idéia da utilização deste ambiente é auxiliar na gerência de programas distribuídos e tolerantes a falhas.

O subsistema Stabilis é responsável pela gerência da informação que descreve a estrutura do subprograma Aplicação. Tais informações são descritas através do modelo estrutural, que consiste de classes e seus relacionamentos. Stabilis mantém a informação estrutural na forma de um esquema[2]. *Esquemas* são compostos por objetos que armazenam informações sobre o modelo de objetos. A partir de um esquema, Stabilis faz a geração parcial do código responsável pelo subprograma Aplicação. Na Figura 2.4, é apresentado o modelo estrutural interno de Stabilis.

[2]Em Inglês: database schema.

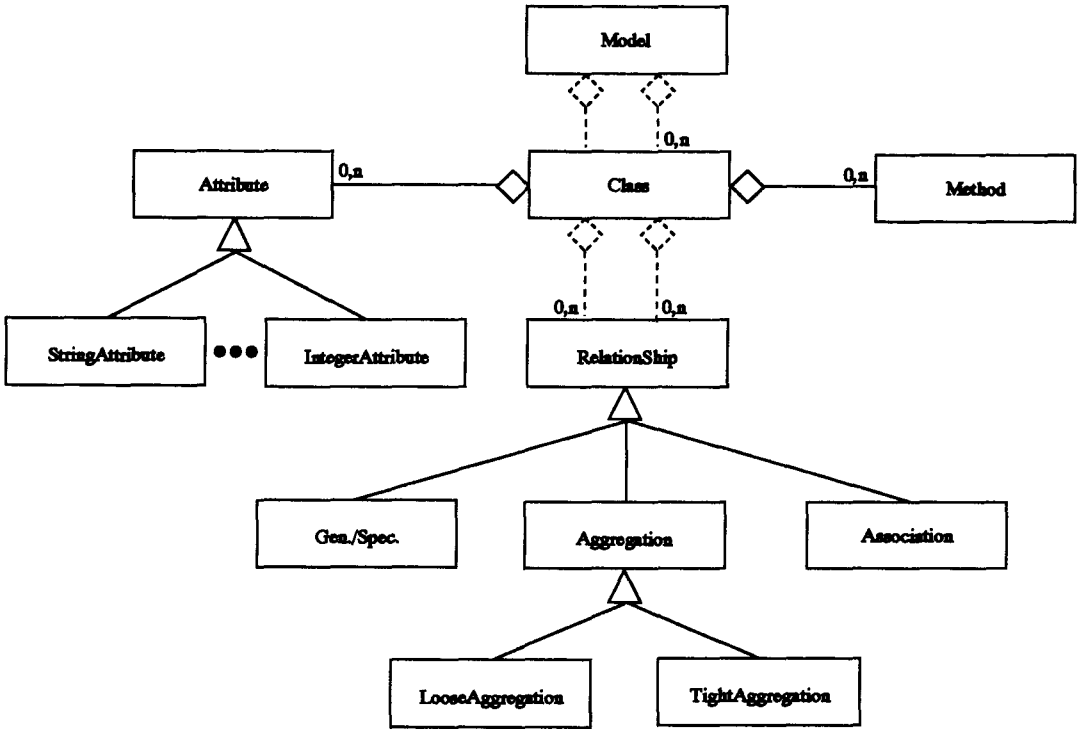



Figura 2.4: Parte do modelo estrutural de Stabilis

Por sua vez, o subsistema Vigil é um núcleo reativo que tem por finalidade cuidar da parte de gerenciamento da informação de controle. Vigil armazena as informações sobre políticas de administração do subprograma Aplicação através de um modelo de controle que também é armazenado como parte do esquema do programa distribuído. Um *Statechart* é utilizado para a representação do subprograma Gerente. *Statecharts* são compostos por estados e transições atômicas entre estados. A Figura 2.5 apresenta o modelo estrutural de Vigil.

É importante ressaltar que a linguagem de programação adotada pelos dois subsiste-

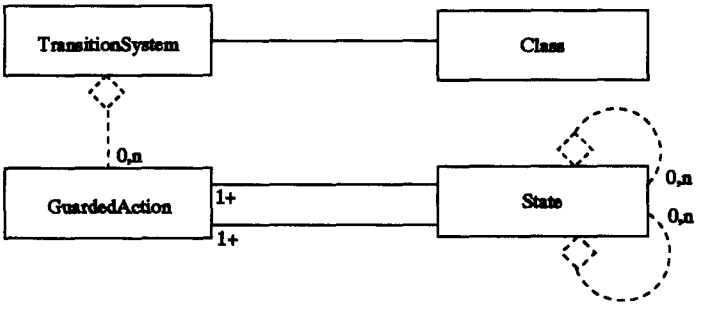



Figura 2.5: Parte do modelo estrutural de Vigil

mas é C++ padrão, ou seja, sem adaptações ou extensões. Consultas realizadas na sua base de dados são efetuadas através de expressões em C++.

A vantagem da utilização deste tipo de ambiente é a abstração de alto nível que ele proporciona ao programador. Durante a implementação de um programa, não há necessidade de se preocupar com vários dos problemas gerados pela distribuição. Em contraste, aspectos de distribuição devem ser considerados quando se trabalha em um nível de abstração mais baixo.

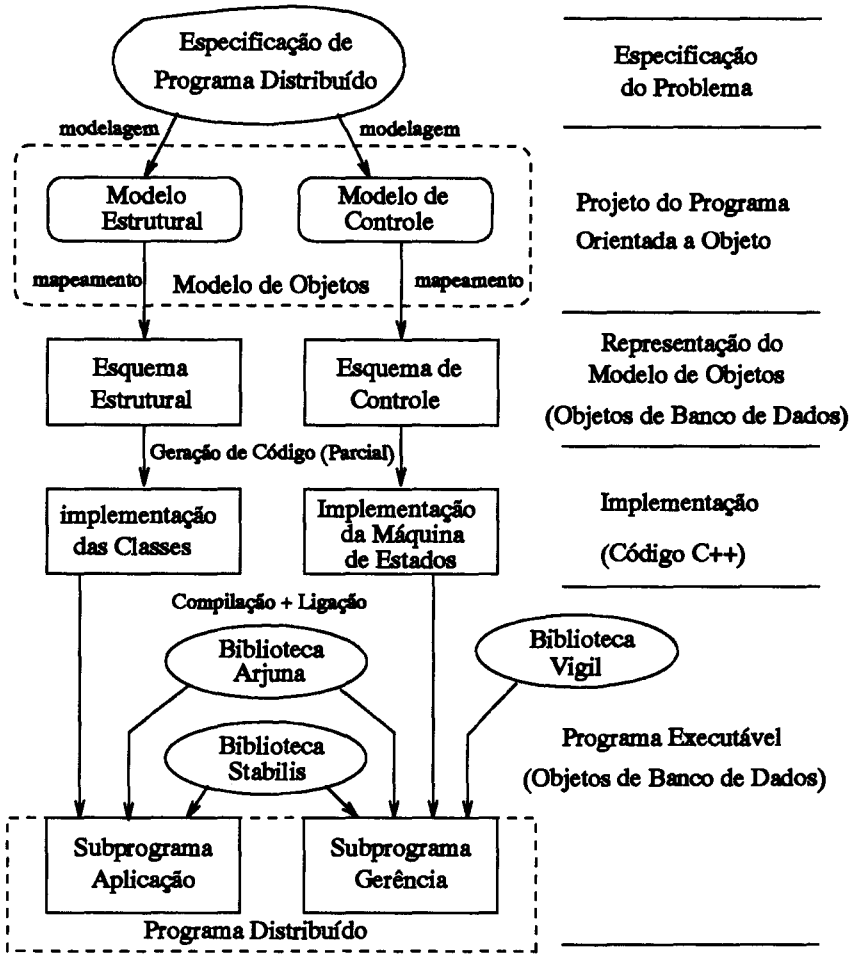



Figura 2.6: Seqüência de passos durante a geração de programas

Para a utilização do ambiente (Figura 2.6), inicialmente o programador deve fazer o levantamento dos requisitos do problema a ser resolvido, ou seja, a análise do problema. Em seguida, baseado nesta análise, deve fazer o projeto de sistema, construindo um modelo de objetos. A partir do modelo de objetos, deve então criar metaprogramas que geram a descrição do modelo de objetos, armazenando-a como esquemas. O modelo estrutural e de controle dão origem, respectivamente, aos esquemas estrutural e de controle. Esses

esquemas são então armazenados como dicionário de dados pelos subsistemas Stabilis e Vigil. Uma vez tendo-se o modelo armazenado no ambiente, o programador deve então criar mais um programa de modo a recuperar o objeto que representa o modelo de objetos na base de dados do ambiente e invocar, a partir dele, o método fornecido pelo ambiente para a geração de código. Uma vez invocado este método, Stabilis, a partir do esquema estrutural, gera parte do código do subprograma Aplicação, enquanto que o subsistema Vigil gera parte do código do subprograma Gerência equivalente ao esquema de controle.

Tendo parte do código gerado, correspondente ao esqueleto do programa distribuído, o programador deve fazer a complementação do código dos subprogramas Aplicação e Gerência, e em seguida compilá-lo e ligá-lo às bibliotecas fornecidas por Stabilis, Vigil e Arjuna, gerando assim o seu código executável. A complementação de código baseia-se basicamente na codificação dos métodos das classes do modelo estrutural.

## 2.2 Arquitetura de Software do Editor

Em geral, ambientes de programação orientada a objetos não incluem os mecanismos necessários para a captura, edição e gerência de modelos de objetos. Neste caso, o programador é responsável pela tradução do modelo de objetos para a linguagem adotada pelo ambiente de programação. Este processo manual de programação dá margem a erros, tanto sintáticos quanto semânticos. Portanto, idealmente, esses ambientes deveriam ser equipados com tais mecanismos. Esses mecanismos minimizam o número de erros de programação através da geração automática de parte do código do programa distribuído. Conseqüentemente, eles reduzem o tempo e custo da programação.

A construção do editor gráfico, doravante nomeado editor, foi motivada pelas razões apresentadas acima. Assim, a integração do editor com o ambiente Stabilis/Vigil traz melhorias significativas para o processo de desenvolvimento de programas distribuídos. Este novo ambiente fornece o suporte necessário à integração das fases de descrição de projeto orientado a objetos, implementação e gerência de programas distribuídos tolerantes a falhas. A sua utilização torna possível ao programador a criação de programas distribuídos a partir da especificação de modelos de objetos desenvolvidos durante as primeiras fases da metodologia orientada a objetos.

Para construir um programa utilizando este novo ambiente é necessário primeiramente fazer a análise do problema e, em seguida, o projeto orientado a objetos. O modelo de objetos gerado nesta última fase, deve ser editado utilizando o editor. Em seguida o programador dispara a geração automática de código. Como resultado, ele obtém os metaprogramas que, quando executados, criam os esquemas do programa distribuído. Com isto, resta apenas ao programador a complementação do código do programa (métodos), que, em seguida, através do editor, é então compilado e ligado às bibliotecas do ambi-

ente de programação. Caso o programador não utilize o editor, todo este processo de mapeamento do modelo de objetos para seus respectivos metaprogramas deve ser realizado manualmente. Como se pode perceber, a utilização do editor simplifica o processo de criação de programas distribuídos no ambiente de programação. Maiores detalhes deste processo pode ser visto no Capítulo 5, onde é apresentado todos os passos para a implementação de um programa exemplo.

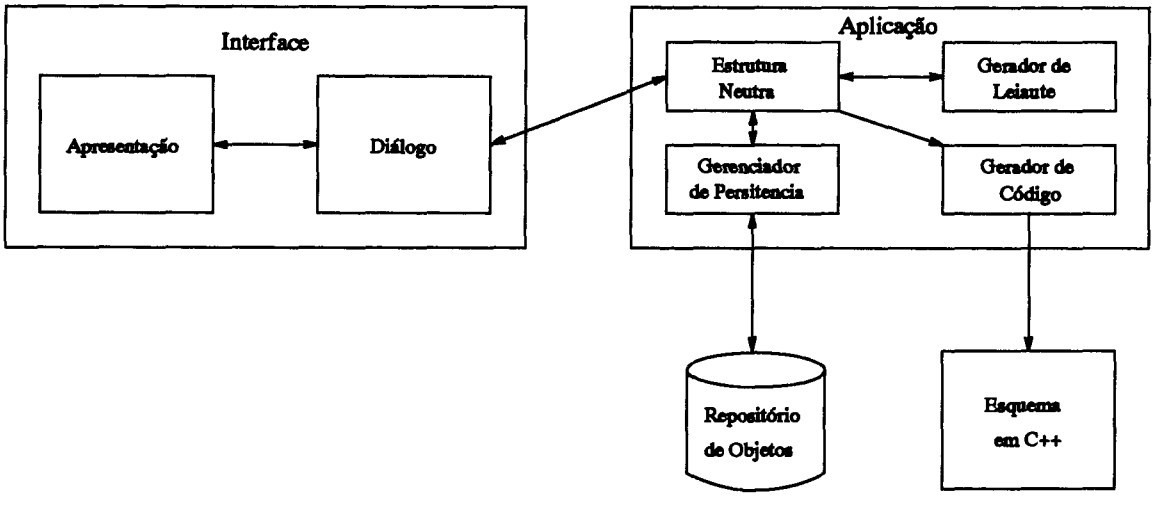



Figura 2.7: Componentes do editor

Dado que o editor é um sistema interativo, sua arquitetura de *software* pode ser dividida logicamente em módulo Interface e módulo Aplicação, sendo que cada um desses módulos é composto de componentes menores (Figura 2.7). Essa divisão lógica visa uma possível obtenção de *independência de diálogo* [31]. Uma vez conseguida, a independência de diálogo permite, entre outras coisas, que possíveis alterações em um dos módulos cause influência limitada no outro (Seção 1.1.1).

O módulo Interface pode ainda ser divido logicamente em dois componentes: *Apresentação* e *Diálogo*. O componente Apresentação é responsável basicamente pela aparência do editor. É através dele que o programador deve interagir durante a construção do modelo de objetos. Já o componente Diálogo é responsável pela comunicação entre Apresentação e Aplicação.

O módulo Aplicação é basicamente divido em quatro componentes:

- **Estrutura Neutra.** Composto por uma estrutura de dados que armazena informações sobre modelo de objetos;
- **Gerador de Leiaute.** Responsável pela geração automática de leiautes para modelos de objetos;

- **Gerador de Código.** Responsável pela geração de código de metaprogramas, que armazenam informações de modelo de objetos em forma de esquemas e do programa que dispara a geração de código no ambiente;
- **Gerenciador de Persistência.** Responsável pelo armazenamento das informações de modelos de objetos em memória estável, bem como pela sua recuperação.

Nos capítulos 3 e 4 são apresentados maiores informações sobre os módulos Interface e Aplicação, respectivamente.

## 2.3 Resumo

Neste Capítulo foi apresentada toda a estrutura modular interna do sistema Arjuna e parte do seu modelo estrutural, através do qual ele torna disponível os conceitos de ações encadeadas, controle de concorrência, recuperação de estado e persistência, que tornam possível construir aplicações distribuídas e tolerantes a falhas. Em seguida, foi apresentada a estrutura do sistema Stabilis/Vigil e, em linhas gerais, a arquitetura do editor gráfico. Além disso, foram apresentados todos os passos envolvidos durante a construção de aplicações no ambiente Stabilis/Vigil, desde a sua especificação até a geração de seu código executável, e como parte deste processo pode ser automatizado quando utiliza-se o editor gráfico.

# Capítulo 3

# Interface

O desenvolvimento de certos tipos de interfaces, como a existente no editor gráfico, envolve, em geral, a utilização de uma metodologia para desenvolvimento de interfaces homem-computador. Estas metodologias têm por objetivo dar suporte necessário para a construção estruturada de interfaces, através da especificação de fases de desenvolvimento bem definidas. Dentre as metodologias existentes, foi escolhida uma cujo processo de desenvolvimento fosse rápido e interativo.

Este capítulo, primeiramente, descreve a metodologia escolhida, mostrando passo por passo cada uma das fases utilizadas durante o processo de desenvolvimento do módulo Interface. Em seguida, apresenta de maneira mais detalhada, cada um dos componentes, que compõe o módulo Interface do editor gráfico.

## 3.1 Metodologia de Construção de Interface Utilizada

Para a construção do módulo de interface utilizou-se como guia a metodologia de Projeto de Interface de Usuário Centrada em Tarefas[1] [37]. A metodologia é organizada em torno de tarefas típicas que usuários devem realizar com o sistema, para o qual a interface está sendo desenvolvida. A premissa básica é que um sistema alcança o sucesso somente se o usuário é capaz de executar as tarefas que necessita realizar de forma conveniente. Ou seja, a ênfase apenas na aplicação ou somente na interface podem comprometer o sucesso do sistema. As tarefas relevantes são identificadas bem no início do processo e são, então, utilizadas para questionar o projeto, bem como auxiliar na tomada de decisões de projeto e na sua posterior avaliação e verificação. A seguir descreve-se as fases utilizadas no desenvolvimento da interface do editor gráfico (Figura 3.1). Contudo, deve ficar claro

[1] Do inglês: *Task-Centered User Interface Design*

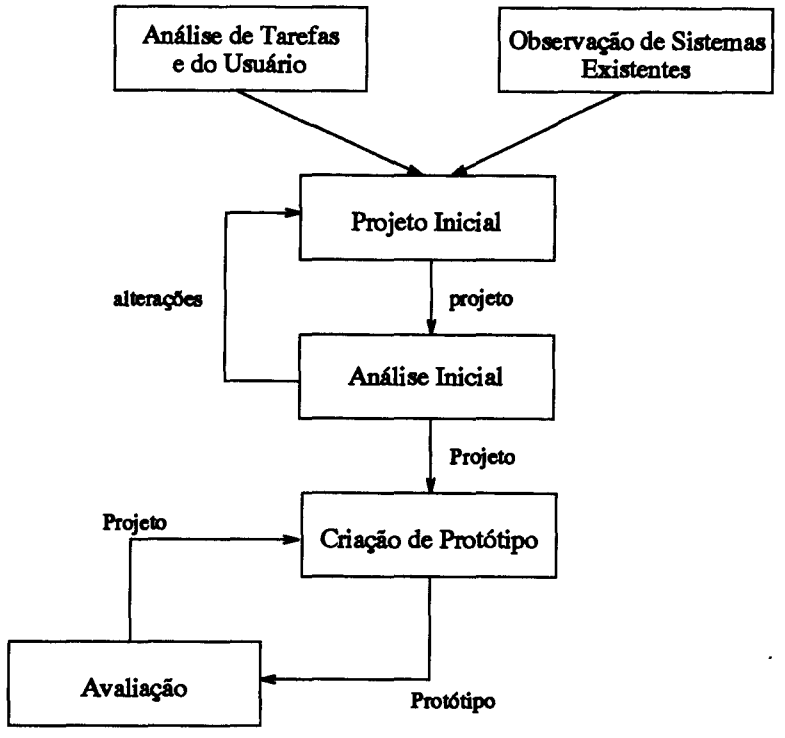



Figura 3.1: Ciclo de vida utilizado na construção da interface

que essas fases tratam-se apenas de um subconjunto das fases propostas pela metodologia original. São elas:

- **Análise de Tarefas e do Usuário.** Nesta fase deve-se identificar as diversas tarefas representativas que o sistema deve realizar, além da identificação dos potenciais usuários do sistema.

  Para o editor gráfico identificou-se como potenciais usuários do sistema programadores com conhecimento em programação distribuída e em orientação a objetos. As tarefas levantadas foram basicamente tarefas associadas com a criação, edição e manipulação de modelo de objetos;

- **Observação de Sistemas Existentes.** Durante esta fase deve-se fazer uma análise em sistemas já existentes, de modo a auxiliar de alguma maneira no desenvolvimento do sistema a ser criado. Prováveis sistemas a serem observados são sistemas já utilizados pelos potenciais usuários e sistemas que possuem algum tipo de relação com o sistema a ser desenvolvido. Este tipo de estudo pode ajudar na definição da aparência e dos tipos de objetos de interação a serem utilizados na interface.

  Foram observados algumas ferramentas e editores gráficos utilizados na criação de modelos de objetos. Alguns destes, no entanto, não contemplam simultaneamente

ambos os modelos estrutural e de controle. São eles:

- With Class;
- Rose;
- State;
- SC;
- BetterState.

Maiores detalhes sobre algumas dos editores e ferramentas acima relacionados podem ser encontrados no Capítulo 6;

- **Projeto Inicial.** Deve-se, nesta fase, criar um esboço do projeto da interface, baseando-se nos estudos realizados nas duas fases anteriores. O esboço não deve ser programado ainda, mas de preferência ser descrito em papel. A codificação do projeto nesta altura do processo, mesmo utilizando ferramentas de prototipação bastante simples, pode forçar o projetista a fazer decisões prematuras que podem comprometer o restante do processo;

  Um esboço inicial do projeto foi elaborado, tentando estabelecer um modelo de apresentação e um possível comportamento para a interface. Esse esboço foi inicialmente feito em papel, tentando levantar os objetos de interação da interface, sua estrutura e leiaute.

- **Análise Inicial.** Nesta fase, as tarefas representativas devem ser usadas para certificar se o sistema está completo. Além disso, deve-se verificar se há funcionalidades extras, que não correspondem a nenhuma das tarefas levantadas.

  Uma análise foi realizada com o intuito de verificar se o modelo de apresentação e comportamento daria o suporte necessário para a realização das tarefas levantadas identificadas;

- **Criação do Protótipo.** Existem dois métodos de prototipação de sistemas [31]. Um dos métodos, chamado *revolucionário*, consiste em criar o protótipo apenas para avaliação do modelo, sendo que protótipo é descartado antes do início da real construção do sistema. Neste método, pode-se utilizar ferramentas próprias para criação de protótipos. Outro método consiste em criar um protótipo e, através de modificações interativas, torná-lo um sistema completo ao final do processo. Esse princípio de prototipação é conhecido como *evolutivo*. Este último método tem a vantagem de ser produtivo, reduzindo o tempo necessário para o desenvolvimento do sistema.

Para criação do protótipo utilizou-se o método evolutivo. De posse das questões levantadas durante o projeto, o protótipo da interface foi sendo gradativamente construído, inserindo, a cada passo da interação, novas funcionalidades equivalentes à cada um das tarefas;

- **Avaliação.** Nesta fase deve-se fazer testes no protótipo de modo a avaliá-lo. Não importa quanta análise tenha sido feita. É importante que se faça testes de modo a verificar se o protótipo provê as funcionalidades adequadas para a execução das tarefas levantadas na primeira fase. Estes testes devem, preferencialmente, ser feitos com pessoas que representem potenciais usuários.

  Para avaliação do protótipo do editor gráfico, foram testadas as funcionalidades inseridas no mesmo, a cada interação, de modo a avaliá-las e a depurá-las. Devido ao curto espaço de tempo e a não disponibilidade de pessoal, foi utilizado o próprio desenvolvedor do protótipo na realização dos testes. Mesmo porque, o desenvolvedor possui os requisitos necessários de modo a ser considerado um potencial usuário do editor. Contudo, provavelmente, poder-se-ia obter melhores resultados nesta fase, caso os testes tivessem sido realizados com outros potenciais usuários e não com o próprio desenvolvedor.

A seção seguinte apresenta cada uma das partes que compõe o componente Apresentação. Todo o desenvolvimento deste componente foi baseado na metodologia aqui apresentada.

## 3.2 Apresentação

O componente Apresentação é responsável pela interação entre o programador e o editor gráfico. Através deste componente, o usuário pode realizar suas tarefas e visualizar seus resultados.

Para a construção da apresentação da interface do editor gráfico utilizou-se vários objetos de interação. Todas as tarefas levantadas durante a fase de análise de tarefas foram associadas a operações disponíveis no editor, que por sua vez foram mapeadas a um ou mais objetos de interação da apresentação.

A apresentação do editor gráfico é composto basicamente por três objetos de interação: Barra de Menu, Barra de Ícones e Área de trabalho. A estrutura básica dos objetos de interação que compõem o editor gráfico pode ser vista na Figura 3.2.

Na Barra de Menu pode-se encontrar todas as operações que podem ser realizadas no editor gráfico. Na Figura 3.3, pode-se observar de forma hierárquica toda a estrutura dos objetos de interação que o compõem. Os diversos objetos de interação são divididos por funcionalidades em: Model, Edit, View, Zoom, Tools, Options e Help.

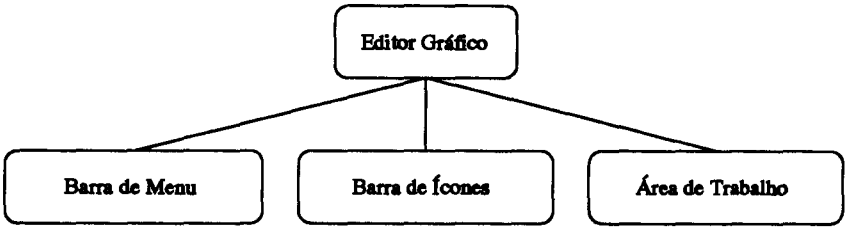



Figura 3.2: Objetos de interação do editor gráfico

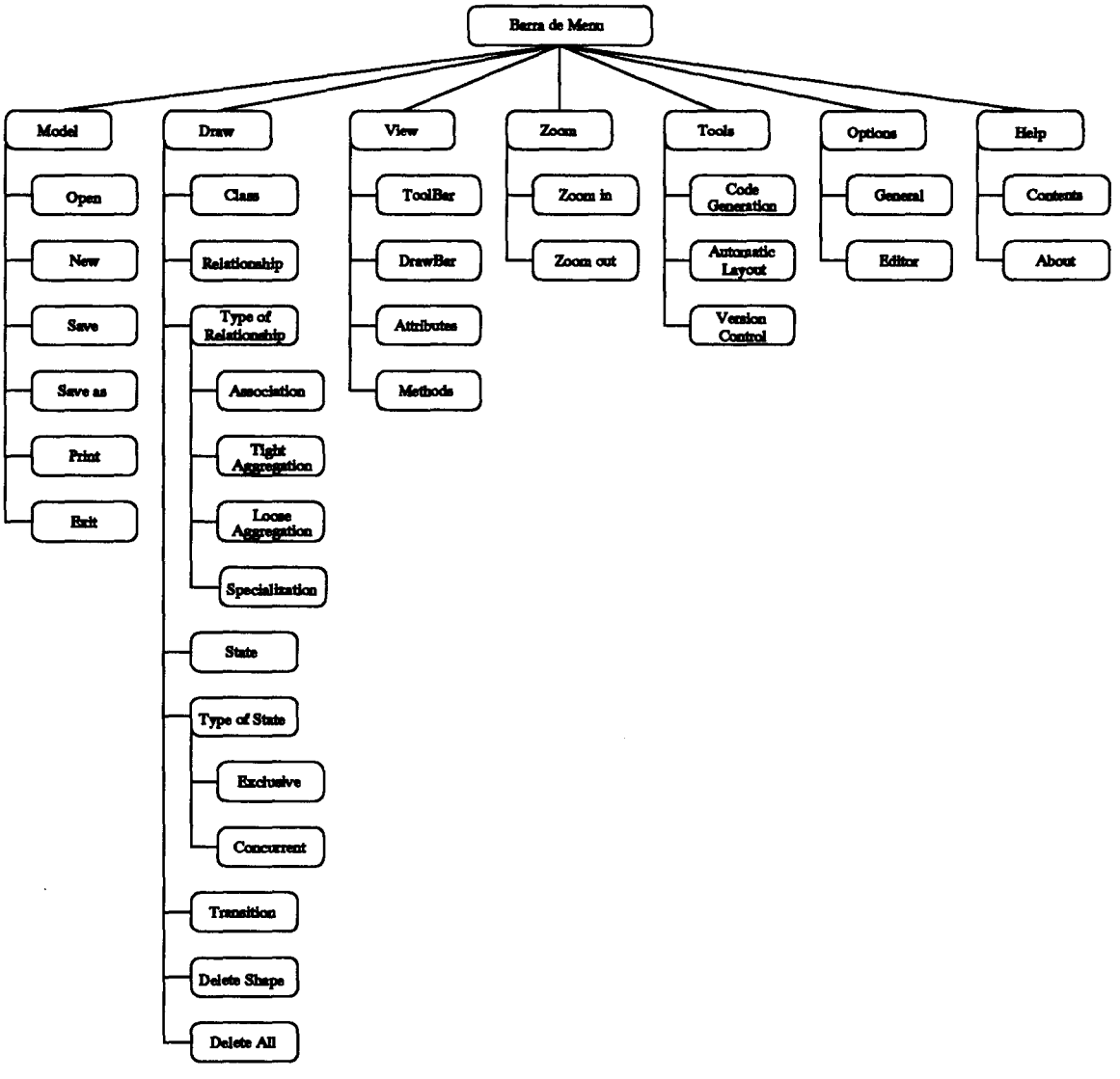



Figura 3.3: Objetos de interação da Barra de Menu

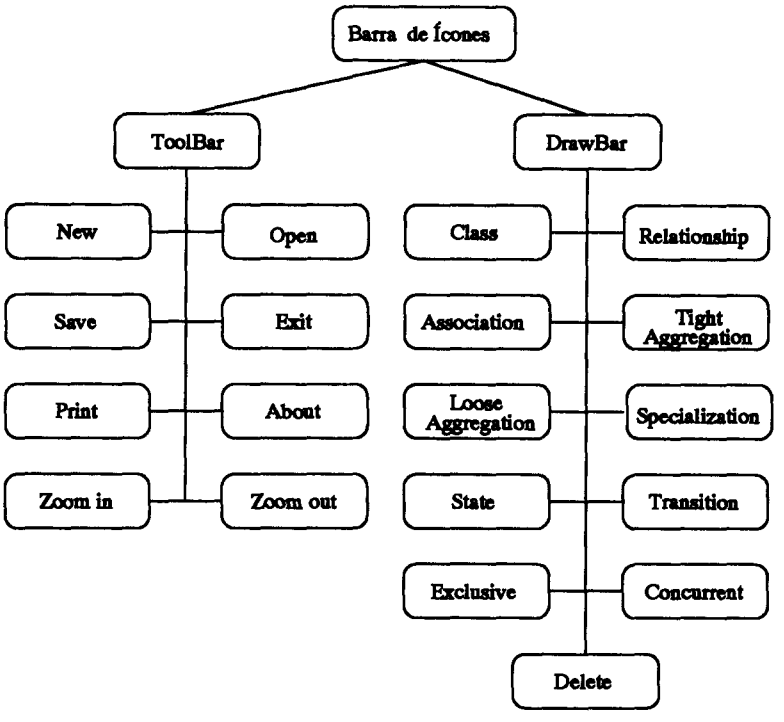



Figura 3.4: Objetos de interação da Barra de Ícone

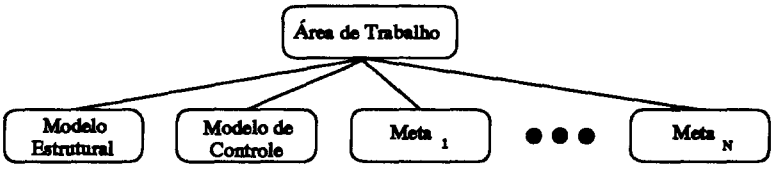



Figura 3.5: Objetos de interação da Área de Trabalho

Todas as operações que podem ser efetuadas na Barra de Ícone também se encontram na Barra de Menu. Estes objetos de interações são utilizados como atalhos, já que o custo (tempo) para acioná-los é menor do que os que compõe a Barra de Menu. Barra de Ícone é funcionalmente divido em ToolBar e DrawBar (Figura 3.4). Em ToolBar encontram-se as operações de manipulação do modelo de objetos e do editor gráfico. Já em DrawBar encontram-se as operações de edição de modelo de objetos.

Toda operação de manipulação direta é realizada no objeto de interação Área de Trabalho, que é subdividido em diversos objetos de interação (Figura 3.5). O objeto de interação Modelo Estrutural é responsável pela interação de modo a permitir a edição e manipulação do modelo estrutural. Semelhantemente, o objeto de interação Modelo de Controle controla a edição e manipulação do modelo de controle. Os objetos de interação $\text{Meta}_1$ a $\text{Meta}_N$ são responsáveis pela edição e manipulação dos diversos metaníveis.

## Implementação

Para a implementação do componente Apresentação optou-se pela utilização de um *toolkit*. *Toolkits* são basicamente bibliotecas de funções que implementam objetos de interação de interfaces gráficas. Em *toolkits*, objetos de interação também são conhecidos como *widgets*.

Baseado em um estudo realizado sobre alguns *toolkits* [35], optou-se pela escolha do *toolkit* Tcl/Tk [41, 62]. Contudo, apesar de Tcl/Tk fornecer um conjunto de objetos de interação razoáveis (por exemplo, janelas, menus, botões etc.), optou-se também pela utilização de um pacote que estende suas funcionalidades, chamado Tix (*Tk Interface Extension*) [36]. Basicamente, Tix oferece novos objetos de interação sendo estes denominados objetos de interação complexo (*mega-widgets*), que são compostos por objetos de interação menores.

Para tentar amenizar o tráfego de comunicação entre Apresentação e Aplicação, uma estrutura de dados foi incorporada no componente Apresentação. Parte desta estrutura é uma cópia da estrutura mantida pelo módulo Aplicação.

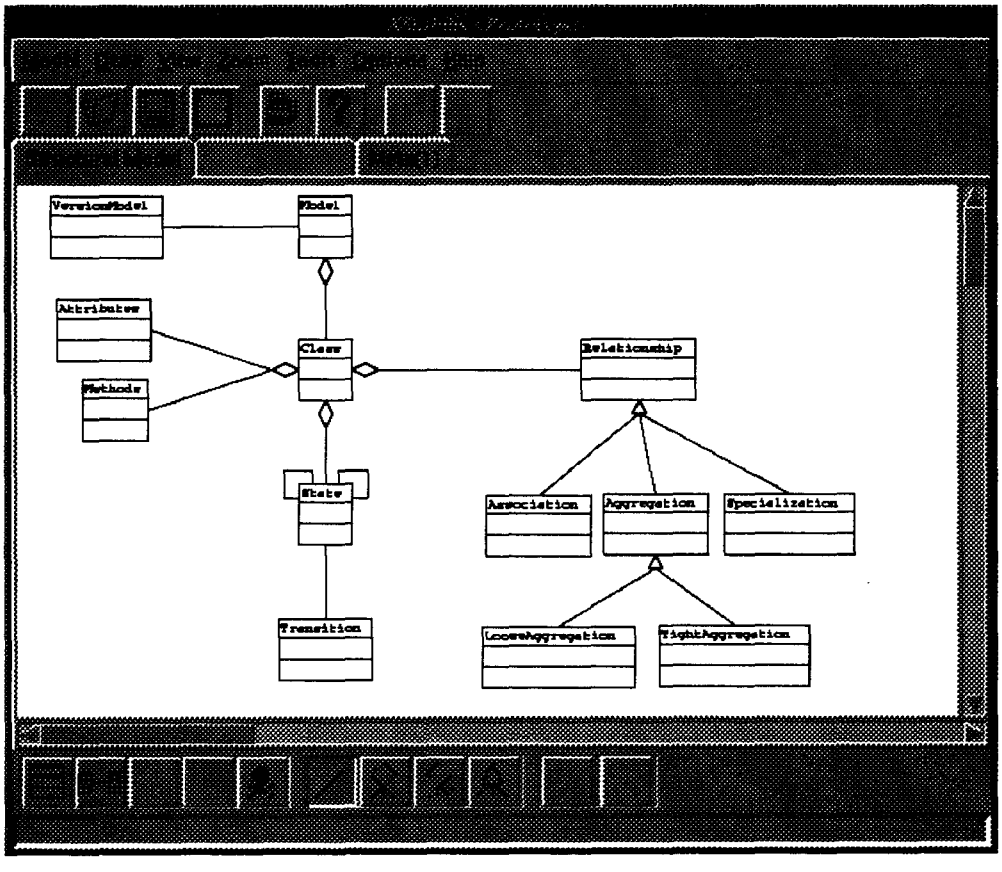



Figura 3.6: Aparência do editor gráfico

Na Figura 3.6 pode-se observar a aparência do componente Apresentação construído

a partir do *toolkit* Tcl/Tk.

## 3.3 Diálogo

A função do componente Diálogo é fazer a comunicação entre o componente Apresentação e o módulo Aplicação. A estrutura deste componente é baseada em rotinas *callbacks*, que são invocadas pelo componente Apresentação. Essas rotinas, uma vez invocadas, fazem as transformações, quando necessárias, dos dados que lhe são passados através de parâmetros e, em seguida, fazem a invocação de métodos de instâncias das classes da estrutura de dados neutra. Ao final do processo, caso a rotina preveja o retorno de algum tipo de dado e dependendo da necessidade, são realizadas transformações nos dados, antes que os mesmo sejam enviados ao componente Aplicação.

A separação entre Apresentação e Aplicação tem como principal objetivo a obtenção de independência de diálogo [31]. A Seção 1.1.1 apresenta algumas das vantagens em se obtendo independência de diálogo.

### Implementação

Dado que o componente Apresentação é implementado em Tcl/Tk [41, 62] e o módulo Aplicação em C++ [20], há a necessidade de fazer a comunicação entre ambos. Tcl/Tk fornece a possibilidade de acoplamento com a linguagem C e, conseqüentemente, com C++. Entretanto, para facilitar este acoplamento utilizou-se um pacote chamado ET (*Embedded Tk*) que, além disso, facilita a criação de programas *stand-alone* baseados em Tcl/Tk.

## 3.4 Resumo

Neste Capítulo, foram descritas a metodologia de construção de interfaces e cada uma das fases utilizadas na criação do módulo Interface. Em seguida, foi apresentada toda a estrutura do componente Apresentação, composta por diferentes objetos, cada um destes é responsável por um tipo de interação com o usuário. Em seguida, foram apresentadas estrutura e objetivo do componente Diálogo.

O Capítulo seguinte apresenta os componentes que compõem o módulo de Aplicação, que é responsável pela parte de armazenamento e processamento de modelo de objetos no editor gráfico.

# Capítulo 4

# Aplicação

Este capítulo é dedicado aos componentes do editor gráfico responsáveis pela manipulação de modelos de objetos. As informações capturadas pelo editor são armazenadas em um formato independente dos formatos utilizados para o traçado dos leiautes e o armazenamento do modelo de objetos no ambiente Stabilis/Vigil. Este formato é denominado estrutura de dados neutra. Cada um dos demais componentes, que possui funcionamento independente uns dos outros, faz referência a esta estrutura de dados para executar suas respectivas funções.

## 4.1 Estrutura de Dados Neutra

A estrutura de dados presente neste componente é responsável pelo armazenamento das informações relativas ao modelo de objetos a ser mantido pelo editor gráfico. Esta estrutura pode ser vista através da Figura 4.1.

Cada uma das classes da estrutura neutra representa um tipo de informação do modelo de objetos desejável de ser armazenada pelo editor. A classe **Model** representa o modelo de objetos em si. Para cada modelo de objetos armazenado no editor deve existir uma instância desta classe. A classe **Class** representa as classes encontradas no modelo de objetos. As classes **Attribute** e **Method** representam, respectivamente, atributos e métodos que cada classe do modelo do modelo de objetos possa possuir. A classe **Relationship** representa os diversos tipos de relações que possam vir a existir entre as classes do modelo de objetos, sendo que as classes **Association**, **Specialization**, **Aggregation**, **TightAggregation** e **LooseAggregation** representam o tipo destes relacionamentos. Já a classe **VersionModel** é utilizada para implementar um mecanismo de controle de versões.

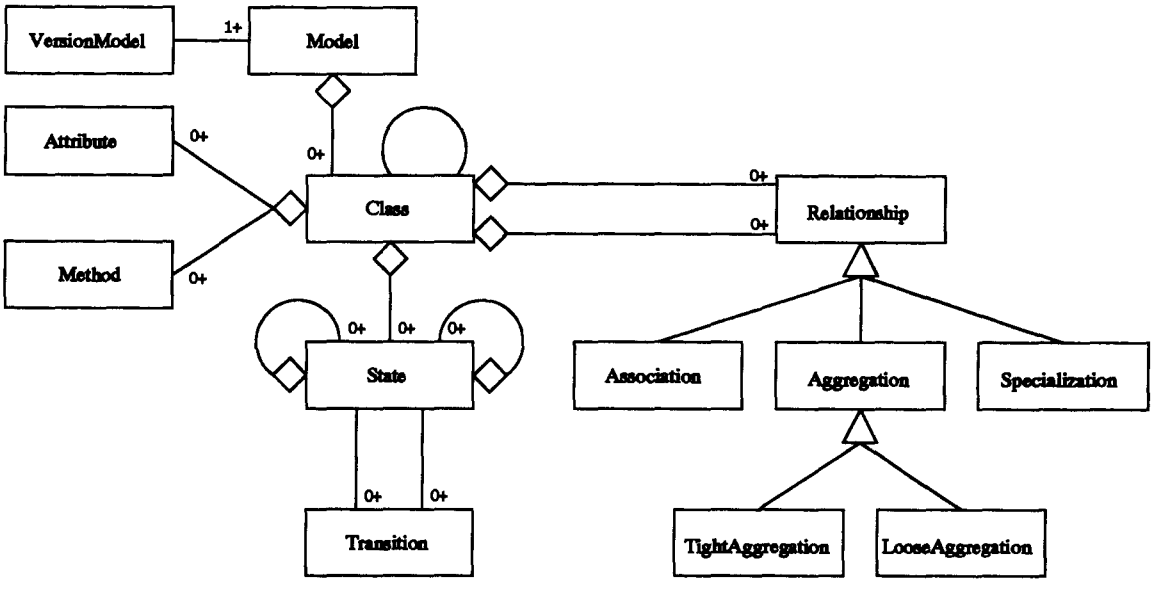



Figura 4.1: Modelo estrutural do componente Estrutura de Dados Neutra

## 4.2 Gerador de Leiaute

Uma visualização pobre de diagramas de modelo de objetos pode dificultar sensivelmente o seu entendimento. Contudo, a edição manual dos diagramas, tendo em vista a tornar mais clara a sua visualização, é uma tarefa cansativa e dispendiosa de tempo. Como esse problema pode ser resolvido, ou pelo menos amenizado, através da implementação de algoritmos que façam a geração automática de leiautes, optou-se então por tornar disponível tal funcionalidade, através da implementação do componente Gerador de Leiaute no editor gráfico. A sua utilização permite que o programador economize tempo, podendo gastá-lo nas demais tarefas envolvidas no desenvolvimento de programas.

### 4.2.1 Algoritmo de Sugiyama e Misue

Dentre os algoritmos analisados (ver Seção 1.3.2), optou-se por escolher o algoritmo de Sugiyama e Misue [53], por dois motivos principais: (i) pelo fato do algoritmo contemplar dígrafos compostos e (ii) por ser um algoritmo largamente utilizado, principalmente o algoritmo para dígrafos desenvolvido por Sugiyama, Tagawa e Toda [56], no qual é baseado. Exemplos de sistemas que implementam pelo menos um desses algoritmos podem ser encontrados em [15, 23, 33, 39, 47, 55].

No algoritmo de Sugiyama e Misue, doravante referenciado apenas como algoritmo de Sugiyama, existem dois tipos de relação entre vértices: inclusão e adjacência. Um grafo composto $D = (V, E, F)$ é obtido pela composição de um grafo de inclusão $D_I = (V, E)$ e

de um grafo de adjacência $D_A = (V, F)$, onde $V$ é um conjunto de vértices, $E$, um conjunto de arestas de inclusão e $F$, um conjunto de arestas de adjacência. Detalhes sobre algumas definições de grafos utilizadas nesta Seção podem ser encontrados na Seção 1.3.1.

Existem, contudo, duas restrições impostas pelo algoritmo em relação ao dígrafo composto $D = (V, E, F)$. São elas:

- O dígrafo de inclusão é necessariamente uma árvore com apenas um vértice raiz;
- Não deve existir relação de adjacência entre vértices ancestrais e descendentes, ou seja, $\{(u, v) \in F | v \in Ancestrais(u) \cup Descendentes(u)\} = \phi$.

O algoritmo define algumas convenções de desenho a serem utilizadas. Para o caso específico deste trabalho, adotou-se as seguintes convenções:

C1. **Retângulo.** Para o modelo estrutural, vértices são desenhados como retângulos, e, para o modelo de controle, são desenhados como retângulos de bordas arredondadas;

C2. **Inclusão.** Uma aresta de inclusão $(u, v)$ é representada de tal maneira que o retângulo correspondendo ao vértice $u$ inclui geometricamente o retângulo correspondendo ao vértice $v$. Retângulos sem relações de inclusão devem ser disjuntos uns dos outros.

C3. **Hierárquico.** Vértices são posicionados hierarquicamente, baseando-se em ambas as relações de inclusão e adjacência, em camadas horizontais encadeadas denominadas níveis compostos.

C4. **Sentido para Baixo** (*Down-arrow*). Tenta-se desenhar todas as arestas do dígrafo em um único sentido. Neste caso específico, de cima para baixo.

Além disso, as seguintes características estéticas são levadas em consideração pelo algoritmo:

R1. **Proximidade** (*Closeness*). Vértices conectados a outros vértices são desenhados tão próximo o quanto for possível;

R2. **Cruzamento entre Linhas** (*Line-crossing*). O número de cruzamentos entre linhas é reduzido o máximo possível;

R3. **Cruzamento de Linhas com Retângulos** (*Line-rect-crossing*). O número de cruzamentos entre arestas de adjacência e vértices é reduzido o máximo possível;

R4. **Linha Reta** (*Line-Straightness*). Arestas de adjacência denominadas "curtas" (arestas que ligam vértices entre níveis compostos adjacentes) são desenhadas como linhas retas e arestas de adjacência denominadas "longas" (arestas que ligam vértices entre níveis compostos não adjacentes) são desenhadas como segmentos de retas, onde reduz-se o número de segmentos de retas e o tamanho da parte vertical dos segmentos é aumentado o máximo possível. Em outras palavras, tende-se a desenhar tais arestas o máximo possível na posição vertical;

R5. **Balanceamento** (*Balancing*). Linhas terminando e originando em um vértice são desenhadas de forma balanceada.

As prioridades entre estas características são especificadas como [53]:

$$R1 > R2 > \cdots > R5 \tag{4.1}$$

Onde $Ri > Rj$ indica que $Ri$ tem prioridade maior que $Rj$.

A essência de todo o algoritmo de Sugiyama é divida em quatro fases:

Fase 1. **Hierarquização** (*Hierarquization*). Consiste em conseguir uma hierarquização para o dígrafo composto, formando assim uma hierarquia composta. Quando isto não for possível, tenta-se a inversão de certas arestas de forma a conseguir a hierarquização. Esta fase tem por objetivo atender as convenções de desenho Inclusão, Hierárquico e Sentido para Baixo;

Fase 2. **Normalização** (*Normalization*). Consiste em converter uma hierarquia composta para um hierarquia composta própria, onde toda aresta de adjacência deve ligar pares de vértices pertencentes ao mesmo nível da hierarquia formada pelo grafo de inclusão. Para tal, deve-se substituir toda aresta de adjacência não-própria por apropriados vértices falsos, arestas de inclusão falsas e arestas de adjacência falsas, de modo a se tornarem próprias. O objetivo desta fase é criar uma rota para as arestas não-adjacentes de modo que elas não cruzem os vértices nos níveis intermediários, obedecendo assim a característica semântica Cruzamento de Linhas com Retângulos;

Fase 3. **Ordenação de Vértices** (*Vertex Ordering*). Consiste em permutar a ordem dos vértices em cada nível da hierarquia de modo a alcançar as características estéticas Proximidade, Cruzamento entre Linhas e Cruzamento de Linhas com Retângulos, o máximo possível;

Fase 4. **Leiaute Métrico** (*Metrical Layout*). O posicionamento horizontal e vertical dos vértices, bem como sua altura e largura, é determinado de modo a obedecer as

características estéticas Proximidade, Linha Reta e Balanceamento. Para o posicionamento horizontal, as ordens dos vértices determinadas no passo anterior devem ser preservadas de modo a manter o número reduzido de cruzamentos entre arestas. Já para o posicionamento vertical, os níveis da hierarquia composta devem ser também preservados.

Na Seção seguinte, é apresentada de forma mais detalhada cada uma destas fases.

```
procedimento Sugiyama(D)
inicio
  DH = Hierarquizacao(D);
  DN = Normalizacao(DH);
  DO = Ordenacao(DN);
  DLM = LeiauteMetrico(DO);
  retorna DLM;
fim
```

## 4.2.2 Descrição das Fases

O algoritmo subentende como entrada um dígrafo composto. Cada uma das fases do algoritmo é executada seqüencialmente, como pode ser visto no trecho de pseudo-código representado através do procedimento **Sugiyama**. O resultado gerado por uma fase é utilizado como entrada para a fase seguinte.

### Hierarquização

Esta fase consiste em transformar o dígrafo composto original em uma hierarquia composta, em vista a satisfazer as convenções de desenho Inclusão e Sentido para Baixo. Para tanto, a cada vértice do dígrafo composto é associado um atributo, denominado *nível composto* ($NCP$), que indica a sua posição na hierarquia composta. Este atributo é formado por uma seqüência de números naturais ($N$) e possui uma ordem lexicográfica bem definida. Por exemplo: (1,1,2) < (1,2) < (1,2,1) < (1,2,2). O tamanho de um $NCP$ de um vértice $v$ é definido pela sua profundidade no dígrafo de inclusão e é denotado por $Tamanhoncp(NCP(v))$, ou seja, $Tamanhoncp(NCP(v)) = Prof(v)$. O i-ésimo número desta seqüência representa a posição do vértice na hierarquia local i, onde i representa a profundidade do vértice na hierarquia de inclusão; hierarquia local é a hierarquia formada pelo conjunto de vértices filhos de seu vértice pai, levando em consideração as arestas de adjacência. Em decorrência do $NCP$, pode-se definir outro atributo denominado *nível*

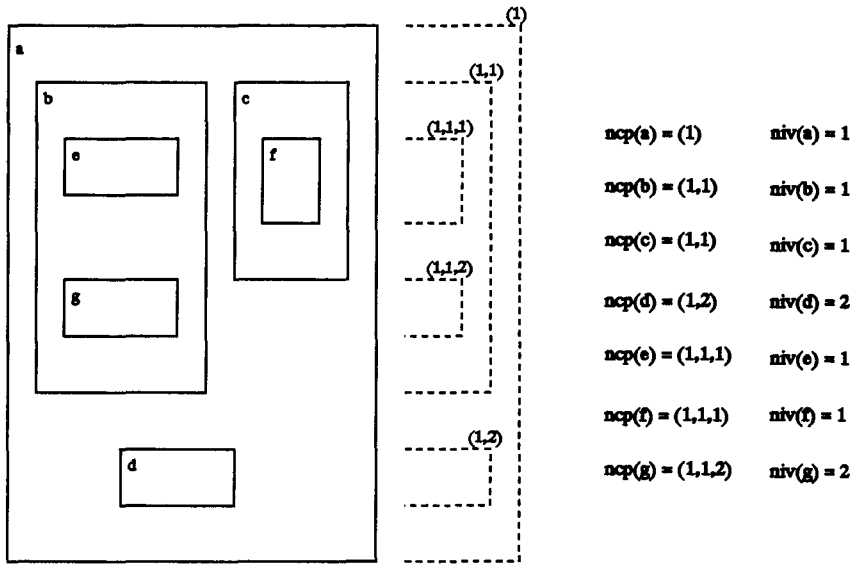



Figura 4.2: Dígrafo composto representado em níveis

($NIV$). O valor deste atributo corresponde ao nível do vértice na hierarquia local a que pertence, ou seja, o valor de $NIV$ corresponde ao último número da seqüência de números do $NCP$. A Figura 4.2 mostra um exemplo de um dígrafo composto desenhado em níveis de acordo com o $NCP$ de cada um de seus vértices.

A partir do $NCP$, a convenção de desenho Inclusão pode ser expressa da seguinte maneira:

$$\forall e = (u, v) \in E,\ NCP(v) = Concatenancp(NCP(u), s) \tag{4.2}$$

Onde $s = NIV(v)$ e $Concatenancp$ é uma função que concatena duas seqüências de números naturais.

Para a formalização da convenção de desenho Sentido para Baixo, são necessárias primeiramente algumas explicações. Para cada aresta de adjacência $e = (u, v)$ do dígrafo composto, existe um único semi-caminho entre os vértices $u$ e $v$ no dígrafo de inclusão, que pode ser expresso como: $p_m(= u), p_{m-1}, \ldots, p_1, t, q_1, \ldots, q_{n-1}, q_n(= v)$, onde $t$ é o vértice topo, ou seja, de menor profundidade neste semi-caminho. Assim, para toda aresta $e = (u, v) \in F$ a convenção de desenho sentido para baixo pode ser expressa da seguinte forma:

$$\begin{aligned} \text{se } Prof(u) > Prof(v) \quad \text{então} \quad & NCP(p_i) \leq NCP(q_i),\ i = 1, \ldots, n-1 \\ & \text{e } NCP(p_n) < NCP(v) \end{aligned} \tag{4.3}$$

$$\begin{aligned} \text{se } Prof(u) \leq Prof(v) \quad \text{então} \quad & NCP(p_i) \leq NCP(q_i),\ i = 1, \ldots, m-1 \\ & \text{e } NCP(u) < NCP(q_m) \end{aligned} \tag{4.4}$$

Ou seja, ela especifica uma ordem lexicográfica entre cada par $(p_i, q_i)$ de vértices de mesma profundidade do semi-caminho de $u$ a $v$, de modo a determinar, através das relações de $\leq$ e $<$, se tais vértices podem ou não pertencer a um mesmo nível na hierarquia composta normalizada.

```
procedimento Hierarquizacao(D)
inicio
  DD = CriaDigrafoCompostoDerivado(D);
  DSC = ResolveCiclo(DD);
  CriaListaVazia(lVertices);
  InsereLista(raiz,lVertices);
  niv(raiz) = 1;
  ncp(raiz) = (1);
  DNC = AssociaNC(lVertices,DSC);
  D = RemoveArestasHierarquizacao(DNC);
  D = VerificaHierarquia(D);
  retorna C;
fim
```

O algoritmo da fase de hierarquização é subdivido em várias sub-fases. A estrutura do algoritmo pode ser vista no trecho de código representado pelo procedimento Hierarquizacao.

```
procedimento CriaDigrafoCompostoDerivado(D)
inicio
  para cada (u,v) pertencente a D faca
    se Prof(u) > Prof(v) entao
      base = Prof(v);
      topo = IguaisAte(Ancestrais(u),Ancestrais(v));
      CriaArestaDerivadaSimples(Ancestral_base(u),v,D);
    senao
      se Prof(u) < Prof(v) entao
        base = Prof(u);
        topo = IguaisAte(Ancestrais(u),Ancestrais(v));
        CriaArestaDerivadaSimples(u,Ancestral_base(v),D);
      senao
        base = Prof(u);
        topo = 1;
```

```
    para i de topo + 1 a base - 1 faca
      CriaArestaDupla(u,v,D);
  retorna D;
fim
```

| | Original | Derivada Simples | Derivada Dupla |
|---|---|---|---|
| Original | Original | Original | Original |
| Derivada Simples | Original | Derivada Simples | Derivada Simples |
| Derivada Dupla | Original | Derivada Simples | Derivada Dupla |

Tabela 4.1: Resolução de conflitos de arestas no dígrafo derivado

O primeiro passo do procedimento Hierarquizacao é a criação, através da invocação do procedimento CriaDigrafoCompostoDerivado, de um dígrafo composto derivado, procurando atender as requisições apresentadas em 4.3 e 4.4. Para tanto, o procedimento propõe-se a substituir toda aresta de adjacência *original*, que não satisfaça tais propriedades, por arestas dentre dois tipos: *derivada simples* e *derivada dupla*. Durante a substituição das arestas, caso haja algum tipo de conflito entre os seus tipos, deve-se seguir as regras de resolução de conflitos apresentadas na Tabela 4.1.

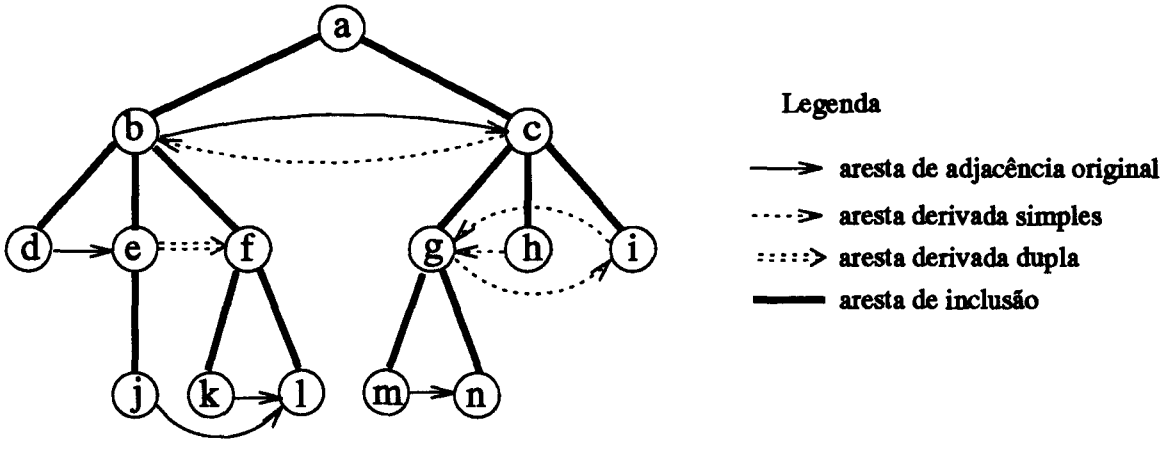



Figura 4.3: Hierarquia de um dígrafo composto

Ao final deste processo, toda aresta de adjacência do dígrafo composto derivado liga vértices de mesma profundidade. A Figura 4.3 mostra um dígrafo composto derivado, baseado no dígrafo da Figura 1.12, onde $(c, b)$, $(g, i)$, $(h, g)$ e $(i, g)$ são arestas derivadas simples e $(e, f)$ é uma aresta derivada dupla.

```
procedimento ResolveCiclo(D)
inicio
  lCFC = ObtemListaCFC(D);
  enquanto nao ListaVazia(lCFC) faca
    para cada cfc em lCFC faca
      e = ObtemFeedBackEdge(cfc);
      se Tipo(e) == ORIGINAL entao
        ReverteAresta(e,D);
      senao
        RemoveAresta(e,D);
    lCFC = ObtemListaCFC(D);
  retorna D;
fim
```

Porém, nem sempre é possível a construção de um hierarquia a partir de um dígrafo composto. Isto, dado a possível existência de ciclos entre as arestas de adjacência do dígrafo composto derivado. Portanto, há a necessidade da remoção de todos os ciclos do dígrafo de adjacência, de modo a torná-los acíclicos. Este problema é conhecido como *minimum feedback edge set* e é reconhecidamente um problema NP-completo [26]. O procedimento ResolveCiclo tenta solucionar este problema através da aplicação de um método heurístico. Este método baseia-se, primeiramente, em encontrar todos os componentes fortemente conexos (cfc) do dígrafo de adjacência. Para cada cfc, escolhe-se uma aresta de adjacência. Se a aresta for do tipo original, então ela é revertida. Caso contrário, ela é removida do dígrafo. Este processo é interativo e termina quando não existir mais nenhum cfc. Algoritmos para o cálculo de cfc's de um grafo podem ser encontrados em [8].

Um dos pontos chave deste processo é a escolha da aresta, pois é desejável reverter (ou remover) apenas um conjunto pequeno de arestas ou até mesmo, quando possível, seu conjunto minimal. Um método heurístico que tenta aumentar a eficiência na procura desse conjunto de arestas pode ser encontrado em [25].

```
procedimeto AssociaNC(lVertices,D)
inicio
  tammaxnivel = AssociaNivelComp(lVertices,D);
  para i de 1 a tammaxnivel faca
    CriaListaVazia(lNiveis$_i$);
    para cada v em lVertices faca
```

```
      se niv(v) == i entao
        InsereLista(v,lNiveis_i);
    se nao ListaVazia(lNiveis_i) entao
      AssociaNC(lNiveis_i,D);
  retorna D;
fim

procedimento AssociaNivelComp(lVertices,D)
inicio
  lNiveis = QuebraEmNiveis(lVertices);
  para v em lNiveis_1 faca
    niv(v) = 1;
    ncp(v) = Concatenancp(ncp(Pai(v),niv(v)));
  para i de 2 a Tamanho(lN) faca
    para cada v em lNiveis_i faca
      para cada e=(u,v) em D faca
        incr = 0;
        se TipoAresta(e) ≠ DUPLA entao
          incr = 1;
        se niv(v) + incr > maxnivel entao
          nivel = niv(v);
          se nivel > maxnivel entao
             maxnivel = nivel;
      niv(v) = nivel;
      ncp(v) = Concatenancp(ncp(Pai(v)),niv(v));
  retorna maxnivel;
fim
```

O passo seguinte é a associação de $NCP$s a cada um dos vértices do dígrafo composto derivado. Sem perda de generalidade, pode-se inicialmente associar ao vértice raiz o $NCP$ de valor um, ou seja, $NCP(raiz) = (1)$. E, a partir daí, associar $NCP$s aos demais vértices do dígrafo. Este processo inicia no vértice raiz e continua recursivamente até os vértices folhas. O cálculo do $NCP$ de um vértice filho é feito baseado em três tipos de informações:

- no $NCP$ de seu vértice pai;
- na sua posição na hierarquia local;
- e nos tipos de aresta que ligam os vértices nesta hierarquia local.

A criação da hierarquia local é realizada pelo procedimento QuebraEmNiveis. Optou-se por não apresentar o seu código devido a sua simplicidade. Para a sua implementação, bastam dois passos. No primeiro passo, deve-se encontrar todos os vértices fonte do dígrafo de adjacência e marcá-los como pertencentes ao primeiro nível. Na segunda fase, deve-se caminhar no dígrafo a partir dos vértices fonte, calculando, para cada vértice, o valor do maior caminho entre um dos vértices fonte até o determinado vértice. Durante o cálculo deste caminho, deve estar claro que o tipo da aresta (original, derivada simples ou derivada dupla) deve ser levado em consideração. Este valor indica a que nível da hierarquia local o vértice pertence.

```
procedimento RemoveArestasHierarquizacao(D)
inicio
  para cada e=(u,v) em D faca
    se TipoAresta(e) == DUPLA ou
       TipoAresta(e) == DERIVADA_SIMPLES entao
      RemoveAresta(e,D);
  retorna D;
fim
```

Em seguida, deve-se remover todas as arestas não originais do dígrafo composto, criadas durante a execução do procedimento CriaDigrafoCompostoDerivado, retornando-o para sua estrutura original. Este passo é realizado através do procedimento RemoveArestasHierarquizacao.

```
procedimento VerificaOrientacao(D)
inicio
  para cada e=(u,v) em D faca
    se nao ncp(u) < ncp(v) entao
       InverteAresta(e,D);
  retorna D;
fim
```

No passo seguinte da fase de hierarquização, deve-se verificar se todas as arestas de adjacência do dígrafo composto obedecem as propriedades 4.3 e 4.4. Todas aquelas que não obedecerem devem ser então revertidas.

Ao final da fase de hierarquização, o dígrafo composto da Figura 1.12 é transformado no dígrafo composto apresentado na Figura 4.4. Neste dígrafo, cada um dos vértices possui uma posição bem definida na hierarquia, de modo a obedecer as convenções de desenho Inclusão e Sentido para Baixo.

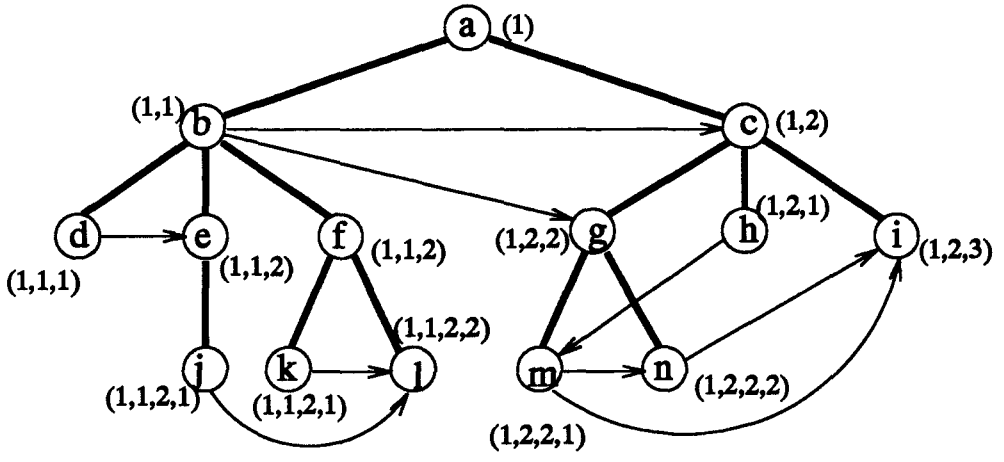



Figura 4.4: Dígrafo composto cujo vértices possuem $NCP$s

### Normalização

Esta fase tem por objetivo transformar um dígrafo composto não próprio em próprio, de modo a obedecer a característica estética Cruzamento de Linhas com Retângulos. Um dígrafo composto é chamado dígrafo composto próprio se e somente se todas as suas arestas de adjacência são ditas próprias. Uma aresta de adjacência $(u, v)$ é dita própria quando obedece a seguinte condição:

$$(u, v) \text{ é própria} \Leftrightarrow NCP(Pai(u)) = NCP(Pai(v)) \text{ e } NIV(u) = NIV(v) - 1 \quad (4.5)$$

Além disso, seja $NCP(u) = (\alpha, s_1, \ldots, s_m)$ e $NCP(v) = (\alpha, t_1, \ldots, t_n)$, onde $\alpha$ é subseqüência comum em ambos os $NCP$s. A partir das propriedades 4.3 e 4.4, pode-se afirmar que $s_1 < t_1$.

Para transformar um dígrafo composto não próprio em próprio, deve-se substituir toda aresta de adjacência $(u, v)$ não própria por adequados vértices falsos, arestas de inclusão falsas e arestas de adjacência falsas. Para tanto, para cada aresta de adjacência não-própria, deve-se aplicar o seguinte método:

$$U \rightarrow (s_1 + 1) \rightarrow \ldots \rightarrow (t_1 - 1) \rightarrow V \quad (4.6)$$

Onde:

$\{(s_1 + 1) \rightarrow \ldots \rightarrow (t_1 - 1)\} \subset \{Pai(u) \text{ ou } Pai(v)\}$

$$U = \begin{cases} u & \text{se } m = 1 \\ v \rightarrow \{\{\ldots\{(s_1, \ldots, s_m + 1) \rightarrow \ldots \rightarrow (s_1, \ldots, M_m)\} \subset (s_1, \ldots, s_{m-1}) \rightarrow \\ \quad (s_1, \ldots, s_{m-1} + 1) \rightarrow \ldots \rightarrow (s_1, \ldots, M_{m-1}) \subset \ldots\} \subset (s_1) & \text{se } m > 1 \end{cases}$$

$$V = \begin{cases} v & \text{se } n = 1 \\ (t_1) \supset \ (t_1,1) \supset (t_1,2) \rightarrow \ldots \rightarrow (t_1,t_2) \supset \ldots \supset & \\ \qquad \{(t_1,\ldots,t_{n-1},1) \rightarrow \ldots \rightarrow (t_1,\ldots,t_{n-1},t_n-1)\}\ldots\} \rightarrow v & \text{se } n > 1 \end{cases}$$

Além disso:

- $(\beta)$ corresponde a um vértice falso com $NCP = (\beta)$;
- $\rightarrow$ corresponde a uma aresta de adjacência falsa;
- $\subset$ e $\supset$ correspondem a arestas de inclusão falsas;
- $M_k$ corresponde ao maior nível da hierarquia local k.

---

```
procedimento Normalizacao(D)
inicio
  para cada e=(u,v) em D faca
    se ncp(Pai(u)) ≠ ncp(Pai(v)) ou
       niv(u) ≠ niv(v) - 1 entao
      RemoveAresta(e,D);
      base = Tamanhoncp(Mesmoncp(ncp(u),ncp(v)));
      pai = Pai(u);
      se Prof(u) > base + 1 entao
        CriaVerticeFalso(w,D);
        CriaArestaInclusaoFalsa(pai,w,D);
        ncp(w) = Faixancp(v,1,base + 1);
      senao
        w = v;
      pai_u = w;
      para i de ncp_base(u) a ncp_base(v) - 1 faca
        CriaVerticeFalso(z,D);
        CriaArestaInclusaoFalsa(pai,z,D);
        CriaArestaAdjacenciaFalsa(w,z,D);
        w = z;
      se Prof(v) > base + 1 entao
        CriaVerticeFalso(z,D);
        CriaArestaInclusaoFalsa(pai,z,D);
        CriaArestaAdjacenciaFalsa(w,z,D);
        ncp(z) = Faixancp(v,1,base + 1);
      senao
```

```
    CriaArestaAdjacenciaFalsa(w,v);
  pai_v = z;
  se Prof(u) > base + 1 entao
    para j de base + 2 a Tamanhoncp(ncp(u)) - 1 faca
      nivelmax = Maximoncp(Faixancp(u,1,j),D);
      CriaVerticeFalso(w,D);
      CriaArestaInclusaoFalsa(pai_u,w,D);
      ncp(w) = Concatenancp(ncp(pai_u),nivelmax);
      para k de nivelmax - 1 a ncp_j(u) faca
        CriaVerticeFalso(z,D);
        CriaArestaAdjacenciaFalsa(z,w,D);
        CriaArestaInclusaoFalsa(pai_u,z,D);
        ncp(z) = Concatenancp(ncp(pai_u),k);
        w = z;
      pai_u = w;
    nivelmax = Maximoncp(Faixa(u,1,j),D)
    se niv(u) == nivelmax entao
      CriaVerticeFalso(w,D);
      CriaArestaInclusaoFalsa(pai_u,w,D);
      ncp(w) = Concatenancp(ncp(pai_u),nivelmax + 1);
    senao
      CriaVerticeFalso(w,D);
      CriaArestaInclusao(pai_u,w,D);
      ncp(w) = Concatenancp(ncp(pai_u),nivelmax);
      para k de nivelmax - 1 a niv(u) faca
        CriaVerticeFalso(z,D);
        CriaArestaAdjacenciaFalsa(z,w,D);
        CriaArestaInclusaoFalsa(pai_u,z,D);
        ncp(z) = Concatenancp(ncp(pai_u),k);
        w = z;
    CriaArestaAdjacenciaFalsa(v,w,D);
  se Prof(v) > base + 1 entao
    para j de base + 2 a Tamanho(ncp(v)) - 1 faca
      nivelmax = Maximoncp(Faixancp(v,1,j),D);
      CriaVerticeFalso(w,D);
      CriaArestaInclusaoFalsa(pai_v,w,D);
      ncp(w) = Concatenancp(ncp(pai_v),1);
      para k de 2 a ncp_j(v) faca
```

```
            CriaVerticeFalso(z,D);
            CriaArestaAdjacenciaFalsa(w,z,D);
            CriaArestaInclusaoFalsa(pai_v,z,D);
            ncp(z) = Concatenancp(ncp(pai_v),k);
            w = z;
          pai_v = w;
        nivelmax = Maximoncp(Faixa(v,1,j),D);
        se niv(v) == 1 entao
          CriaVerticeFalso(w,D);
          CriaArestaInclusaoFalsa(pai_v,w,D);
          ncp(w) = Concatenancp(ncp(pai_v),0);
        senao
          CriaVerticeFalso(w,D);
          CriaArestaInclusaoFalsa(pai_v,w,D);
          ncp(w) = Concatenancp(ncp(pai_v),1);
          para k de 2 a niv(v) faca
            CriaVerticeFalso(z,D);
            CriaArestaAdjacenciaFalsa(w,z,D);
            CriaArestaInclusaoFalsa(pai_v,z,D);
            ncp(z) = Concatenancp(ncp(pai_v),k);
            w = z;
        CriaArestaAdjacenciaFalsa(w,v,D);
  retorna D;
fim
```

Este método é implementado através do procedimento Normalizacao. Todo o processo envolvido neste método pode ser divido em três etapas. Na primeira etapa, caso $m > 1$, deve-se subir na hierarquia a partir da profundidade do vértice $u$, que se encontra no nível $m$, até a profundidade $Prof(\alpha) + 1$, criando-se vértices falsos e interligando-os à hierarquia de inclusão através de arestas de inclusão falsas. Além disso, durante este processo, a cada nível da hierarquia, deve-se criar vértices falsos a partir de $s_i$ até o nível máximo ($M_i$), ligando-os através de arestas de adjacências falsas.

Numa segunda etapa, deve-se criar vértices falsos de $s_1+1$ até $t_1-1$, ligando-os através de arestas de adjacência falsas. Além disso, todos estes vértices falsos devem ser ligados, através de arestas de inclusão falsas, a um vértice pai, que pode ser tanto o vértice pai de $u$ quanto o de $v$.

Já na terceira fase, caso $n > 1$, inversamente ao processo da primeira fase, deve-se descer na hierarquia a partir do último vértice falso criado na fase anterior até a profundidade $n$, criando vértices falsos e interligando-os à hierarquia de inclusão através

de arestas de inclusão falsa. Concorrentemente, a cada nível da hierarquia, deve-se criar vértices falsos a partir de $t_i$ até o nível máximo ($M_i$), ligando-os através de arestas de adjacência falsas.

Além disso, deve-se verificar se $s_m$ é igual a $M_m$. Se isto ocorrer, um nível virtual $M_m + 1$ deve ser adicionado à hierarquia local $m$. Semelhantemente, deve-se verificar também se $t_n$ é igual a 1. Neste caso, um nível virtual zero deve ser adicionado à hierarquia local n.

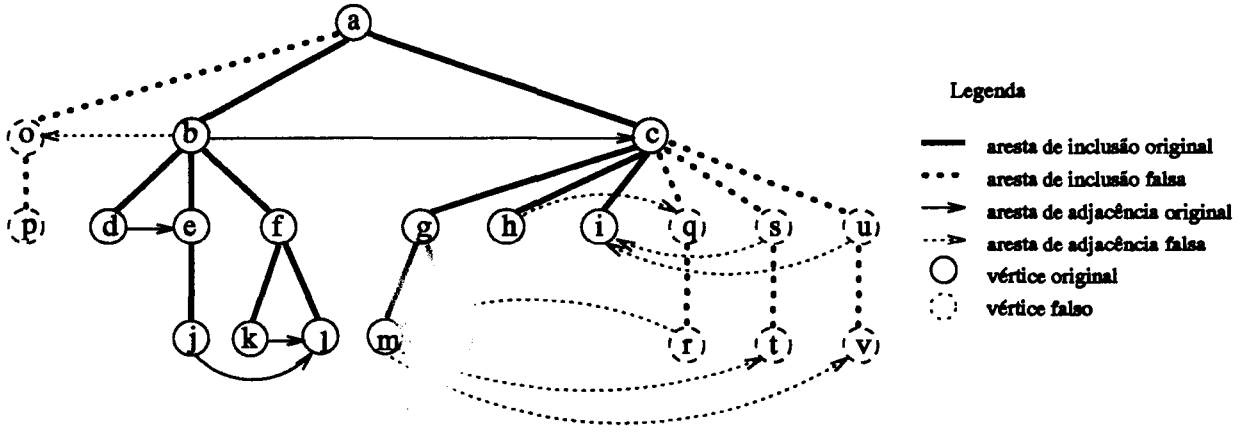



Figura 4.5: Dígrafo composto próprio

No dígrafo composto apresentado na Figura 4.4, as arestas $(b, g)$, $(h, m)$, $(m, i)$ e $(n, i)$ são ditas não próprias. A Figura 4.5 mostra o dígrafo composto próprio gerado após a aplicação da fase de normalização sobre o dígrafo da Figura 4.4.

Em resumo, o objetivo desta fase é criar uma rota para as arestas não-adjacentes de modo que elas não cruzem os vértices nos níveis intermediários, em vistas a obedecer a característica semântica Cruzamento de Linhas com Retângulos. Através da Figura 4.6 pode-se observar melhor como todo esse processo é realizado. No dígrafo apresentado na Figura 4.6a, a aresta $(b, e)$ é uma aresta não-própria. A Figura 4.6b mostra o dígrafo composto próprio obtido a partir do dígrafo não próprio da Figura 4.6a, onde a aresta não-própria $(b, e)$ é substituída pelos vértices falsos $f$ e $g$, pelas arestas de inclusão falsas $(a, f)$ e $(f, g)$ e pelas arestas de adjacência falsas $(b, f)$ e $(g, e)$. A Figura 4.6c apresenta o leiaute do dígrafo composto próprio, com cada vértice disposto de acordo com seu $NCP$. E finalmente, a Figura 4.6d mostra o leiaute final, juntamente com o roteamento das arestas não-próprias.

### Ordenação

Esta fase tem por objetivo tornar um dígrafo composto próprio em um dígrafo composto ordenado, visando atender as seguintes características estéticas: Proximidade, Cruza-

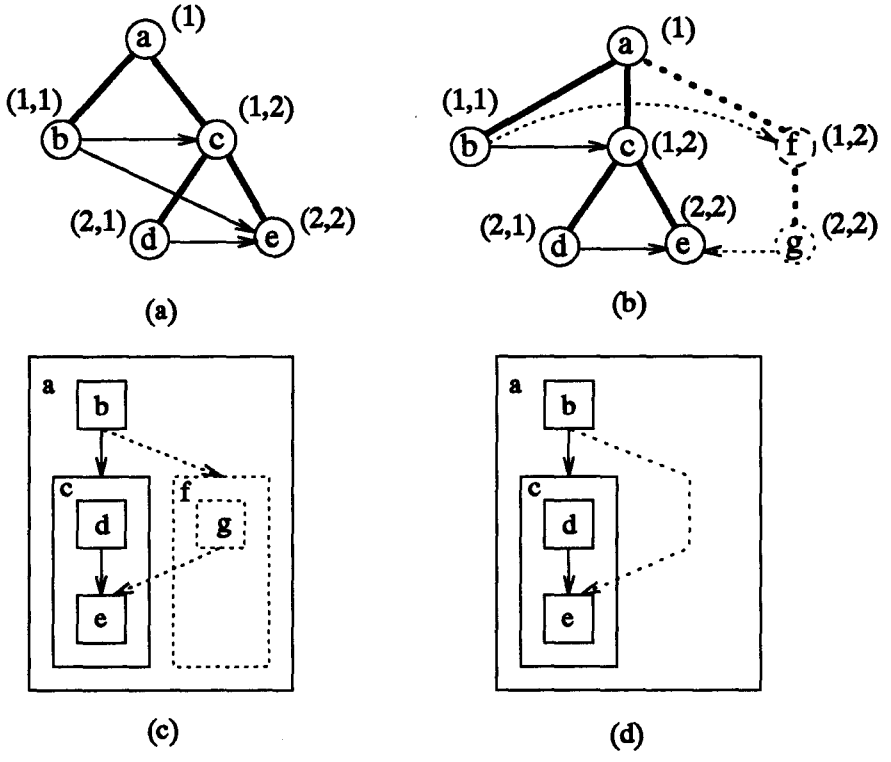



Figura 4.6: Roteamento de arestas não próprias

mento entre Linhas e Cruzamento de Linhas com Retângulos.

Antes da apresentação do método de ordenação de um dígrafo próprio, é necessária a apresentação de algumas definições. Suponha que, para todo vértice não folha $v$ do dígrafo composto próprio, pode-se particionar o seu conjunto vértices filhos da seguinte maneira:

$$Filhos(v) = N_1(v) \cup \ldots \cup N_i(v) \cup \ldots \cup N_{n(v)}(v) \tag{4.7}$$

Onde $N_i(v) = \{u \in Filhos(v) | NIV(u) = i\}$, $N_i(v)$ é chamado i-ésimo nível e $n(v)$ é o nível máximo entre os vértices filhos de $v$.

Baseado nesta definição pode-se definir uma ordem $\sigma$ em todo o dígrafo composto, onde $\sigma = (\sigma(v_1), \ldots, \sigma(v_N))$, $\sigma(v_j) = (\sigma_1(v_j), \ldots, \sigma_{n(v_j)}(v_j))$ e $\sigma_i(v_j)$ é uma ordem entre todos os vértices de cada $N_i(v_j)$. Por exemplo, a ordem do dígrafo da Figura 4.7 é dado por $\sigma = (((a)), ((b,c,d,e)), ((f),(k)), ((g,h),(l,m,n)), ((i),(o),(q)), ((j),(p),(r)), ((s)), ((t),(u)), ((v)), ((x,y),(z)))$.

Agora, suponha que $A(v)$ seja o conjunto de todos os vértices que possuam o $NCP$ igual ao de $v$, excetuando-se o próprio vértice $v$ (ou seja, $A(v) = \{u \in V | NCP(u) = NCP(v)\} - \{v\}$). Tal conjunto $A(v)$ pode ser particionado em dois subconjuntos: $A^E(v)$ e $A^D(v)$. Onde $A^D(v)$ e $A^E(v)$ são compostos pelos vértices a esquerda e a direita de $v$

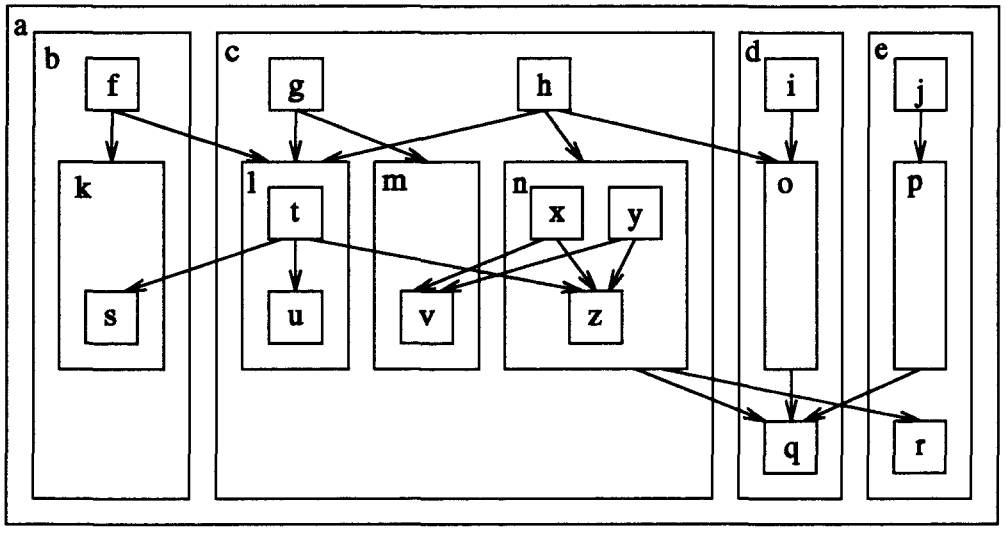



Figura 4.7: Exemplo de dígrafo composto ordenado

na ordem $\sigma$, respectivamente. Por exemplo, na Figura 4.7, $A(c) = \{b, d, e\}$, $A^E(c) = \{b\}$ e $A^D(c) = \{d, e\}$.

Para cada vértice não folha $v$, uma hierarquia local pode ser definida como sendo formada por $Filhos(v)$ obedecendo a propriedade 4.7. Além disso, pode-se definir um conjunto de arestas de adjacência $F(v)$ desta hierarquia local como sendo formada por dois subconjuntos: $F^d$ e $F^m$.

$F^d$ é o conjunto de arestas de adjacência que ligam vértices de níveis diferentes na hierarquia local, ou seja, $F^d = \{(u, w) \in F | u, w \in Filhos(v)\}$. O fato de se garantir que toda aresta de $F^d$ liga necessariamente vértices de níveis diferentes é dado pelas propriedades 4.3, 4.4 e 4.7. As duas primeiras propriedades garantem que toda aresta de adjacência liga vértices com diferentes $NCP$ e a última garante que, em conseqüência disto, eles estarão em níveis diferentes. Na Figura 4.7, $F^d(c) = \{(g, l), (g, m), (h, l), (h, n)\}$.

$F^m$ é definido como sendo o conjunto de arestas de adjacência entre vértices de mesmo nível, ou seja, $F^m = \{(u, w) | (x, y) \in F, x \in Descendentes(u) - \{u\}, y \in Descendentes(w) - \{w\}, u, w \in Filhos(v)\}$. A garantia de que $u$ e $w$ são vértices que pertencem a um mesmo nível é dada através da seguinte demonstração por contradição: suponha que $u$ e $w$ sejam de níveis diferentes com $NCP(u) = (\alpha, s_1)$ e $NCP(w) = (\alpha, t_1)$, onde $s_1 \neq t_1$ já que eles pertencem a níveis diferentes. Agora suponha que $x$ seja filho de $u$, que $y$ seja filho de $w$, que exista uma aresta de adjacência $(x, y)$ e que $NCP(x) = (\alpha, s_1, s_2)$ e $NCP(y) = (\alpha, t_1, t_2)$. Isto é uma contradição, já que a aresta $(x, y)$ não é própria, pois não obedece a propriedade 4.5, dado que os pais de $x$ e $y$, respectivamente $u$ e $w$, possuem $NCP$s diferentes. Na Figura 4.7, $F^m(c) = \{(l, n), (n, m)\}$.

Além do conjunto de arestas $F$, para cada vértice $u \in Filhos(v)$, pode-se definir duas

características denominadas $\lambda(u)$ e $\rho(u)$, que representam o números de arestas entre descendentes de $u$ e descendentes de $A^E(v)$ e $A^D(v)$, respectivamente. Por exemplo, na Figura 4.7, $\lambda(l) = 2$ (devido as arestas $(f,l)$ e $(t,s)$), $\rho(l) = 0$, $\lambda(n) = 0$ e $\rho(n) = 2$ (devido as arestas $(n,1)$ e $(n,r)$).

```
procedimento Ordenacao()
inicio
  OrdenacaoGlobal(raiz);
fim

procedimento OrdenacaoGlobal(v)
inicio
  V = Splitting(Filhos(v));
  lNiveis = QuebraEmNiveis(V);
  OrdenacaoLocal(lNiveis);
  para cada u em Filhos(v) faca
    OrdenacaoGlobal(u);
fim
```

Uma vez apresentadas as definições acima, pode-se passar para o problema de criação do dígrafo composto ordenado. Este problema pode ser dividido em três subproblemas, cada um relacionado a uma das características semânticas: Proximidade (SP1), Cruzamento entre Linhas (SP2) e Cruzamento de Linhas com Retângulos (SP3). Para a resolução destes subproblemas, o algoritmo estabelece a seguinte prioridade elaborada empiricamente:

$$SP1 > SP2 > SP3 \tag{4.8}$$

Onde $SPi > SPj$ indica que $SPi$ tem prioridade maior que $SPj$.

```
procedimento Splitting(Filhos(v))
inicio
  para cada u em Filhos(v) faca
    delta = ro(v) - lambda(v);
    se delta > 0 entao
      ''Fixa v na posicao esquerda''
      Remove(v,N);
    senao se delta < 0 entao
      ''Fixa v na posicao direita''
      Remove(v,N);
```

```
  retorna N;
fim
```

Para solucionar o subproblema associado à característica estética Proximidade, pode-se aplicar um método chamado *Splitting*. Este método, que pode visto através do procedimento **Splitting**, baseia-se em um procedimento heurístico onde, para cada vértice $u$, calcula-se $\Delta = \lambda(u) - \rho(u)$. Se $\Delta > 0$ ou $\Delta < 0$, o vértice $u$ é colocado na posição mais a esquerda ou mais a direita, respectivamente, na ordem do nível a que pertence ($\sigma$). Caso exista mais de um vértice nesta condição, então o vértice com maior valor de $|\Delta|$ é colocado mais próximo as extremidades da ordem.

```
procedimento OrdenacaoLocal(lNiveis)
inicio
  n = Tamanho(lNiveis);
  para i de 1 a CONSTANTE_HEURISTICA faca
    MetodoBaricentroInsercaoCima(lNiveis_1);
    para j de 2 a n faca
      MetodoBaricentroCima(lNiveis_j);
      MetodoBaricentroInsercaoCima(lNiveis_j);
    MetodoBaricentroInsercaoBaixo(lNiveis_n);
    para j de n - 1 a 1 faca
      MetodoBaricentroBaixo(lNiveis_j);
      MetodoBaricentroInsercaoBaixo(lNiveis_j);
fim
```

O subproblema associado à característica estética Cruzamento entre Linhas pode ser solucionado através de um método heurístico denominado método *Baricentro*, que é subdividido em dois métodos: *Baricentro Cima* e *Baricentro Baixo*. Tais métodos baseiam-se, inicialmente, no cálculo do valor baricentro cima ($B_c$) e baricentro baixo ($B_b$) de cada vértice. Os valores dos baricentros de um vértice $v$ podem ser calculados através das seguintes fórmulas:

$$B_c(v) = \begin{cases} \frac{\sum_{(u,v) \in Fc(v)} ordem(u)}{|Fc(v)|} & \text{se } |Fc(v)| > 0 \\ 0 & \text{caso contrário} \end{cases} \tag{4.9}$$

$$B_b(v) = \begin{cases} \frac{\sum_{(v,u) \in Fd(v)} ordem(u)}{|Fd(v)|} & \text{se } |Fd(v)| > 0 \\ 0 & \text{caso contrário} \end{cases} \tag{4.10}$$

Onde, $v \in V$, $Fc(v) \subset F^d$ é o conjunto de arestas convergentes a $v$, $Fd(v) \subset F^d$ é o conjunto de arestas divergentes a $v$ e $ordem(u)$ é posição do vértice $u$ na ordem do nível a que ele pertence.

Uma vez calculado o baricentro ($B_c$ ou $B_b$), deve-se então ordenar os vértices em ordem crescente de acordo com este valor.

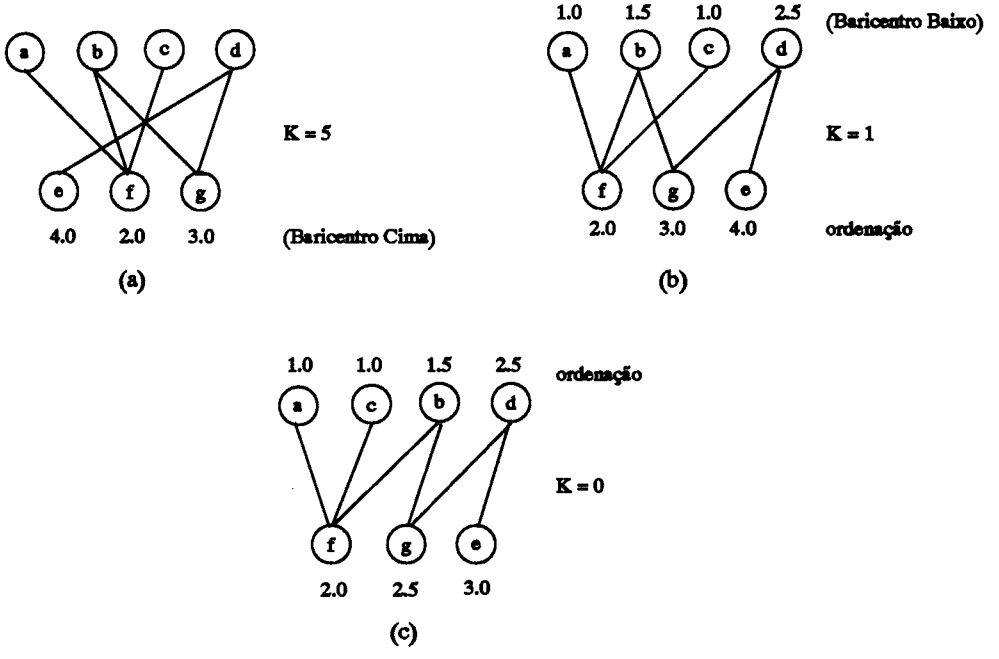



Figura 4.8: Aplicação do método Baricentro em uma hierarquia de 2 níveis

A Figura 4.8 mostra a aplicação do método sobre um dígrafo de apenas dois níveis. Durante a evolução do método (Figuras 4.8a-c), pode-se observar claramente a sua eficiência na redução do número de cruzamento entre linhas, indicado através variável K. As orientações das arestas de adjacência (de cima para baixo) foram omitidas de modo a melhorar a clareza do desenho.

Com relação ao subproblema associado a característica estética Cruzamento de Linhas com Retângulos, há um método heurístico chamado método *Baricentro de Inserção* que procura solucionar esse subproblema. Semelhantemente ao método *Baricentro*, este método pode ser dividido em método *Baricentro Inserção Cima* e método *Baricentro Inserção Baixo*. Estes métodos consistem em inserir vértices temporários entre todas arestas de $F^m$. Estes vértices temporários formam um novo nível (em cima, no caso do Baricentro Inserção Cima ou embaixo, no caso do Baricentro Inserção Baixo). Assim, juntamente com o nível original, tem-se dois níveis de hierarquia. Com isto, pode-se utilizar o *Método Baricentro* como descrito no subproblema anterior.

Para ilustrar melhor o método, observe a aplicação do *Método Baricentro Inserção*

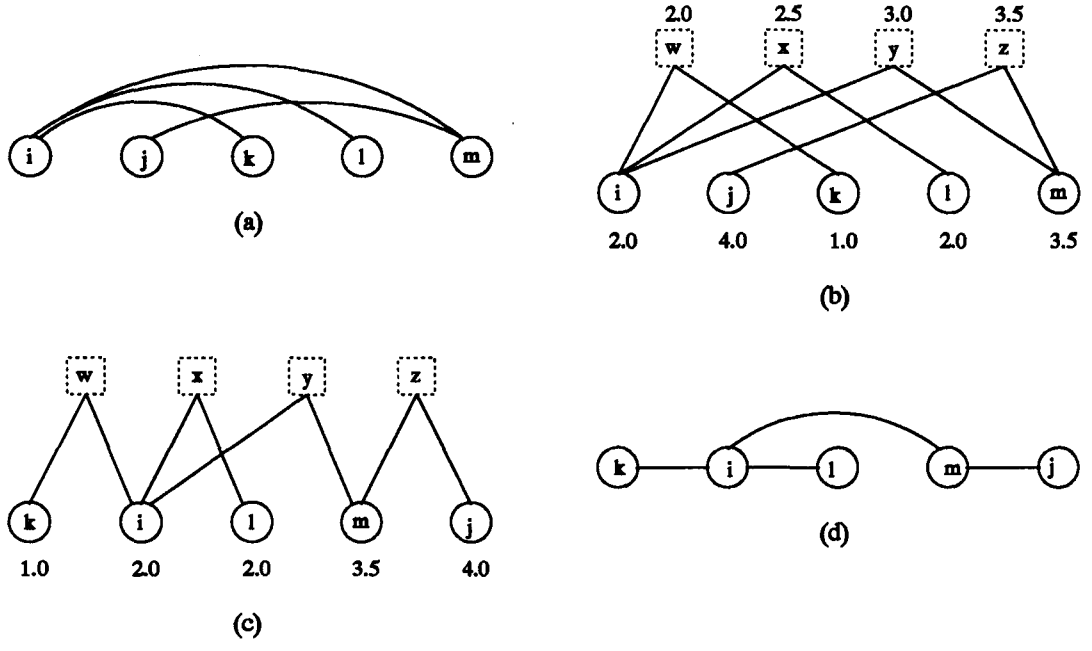



Figura 4.9: Aplicação do método Baricentro de Inserção em uma hierarquia de 2 níveis

*Cima* na Figura 4.9. Primeiramente, a partir do dígrafo da Figura 4.9a faz-se a inserção dos vértices temporários $w$, $x$, $y$ e $z$ (Figura 4.9b). Logo em seguida, aplica-se o *Método Baricentro Cima* ao segundo nível (nível original), que calcula o valor do baricentro ($B_c$) de cada um dos vértices (Figura 4.9b) e em seguida ordena-os, resultando no dígrafo da Figura 4.9c. O resultado final do método pode ser visto na Figura 4.9d. Dado que as orientações não afetam a essência do método e na tentativa de melhorar a clareza do dígrafo, as orientações das arestas do dígrafo da Figura 4.9 não são apresentadas.

O processo envolvido nos métodos *Baricentro* e *Baricentro Inserção* é interativo, começando do primeiro nível e indo até o último nível, intercalando suas aplicações. Este processo é repetido um número fixo ($N$) de vezes. O valor desse número é empírico. Valores considerados "razoáveis" podem ser encontrado após realizações de testes.

```
procedimento LeiauteMetrico()
inicio
  CriaListaVazia(lVertices);
  InsereLista(raiz,lVertices);
  LeiauteMetricoHorizontal(lVertices);
  LeiauteMetricoVertical(lVertices);
  LeiauteGlobal(lVertices);
fim
```

### Leiaute Métrico

Esta fase tem como objetivo construir um leiaute métrico a partir de um dígrafo composto ordenado de modo a atender as características estéticas Proximidade, Linha Reta e Balanceamento. Para tanto, nesta seção apresenta-se um método heurístico que se baseia basicamente em dois problemas: posicionamento de vértices (isto é, determinação das posições horizontais e verticais, além da determinação da altura e largura dos retângulos) e roteamento de arestas. O código para implementação desta fase pode ser visto através do procedimento LeiauteMetrico.

O problema da definição das rotas é mais simples do que o primeiro. Para cada aresta de adjacência do dígrafo, dependendo se for própria ou não, aplica-se o seguinte método para a obtenção de sua rota:

- **Aresta de adjacência própria.** É obtida através de um único segmento de reta ligando os vértices;
- **Aresta de adjacência não própria.** É obtida através de vários segmentos de reta, onde os pontos de junção dos segmentos é dada pelas posições dos vértices falsos criados durante a fase de normalização (Figura 4.6c-d).

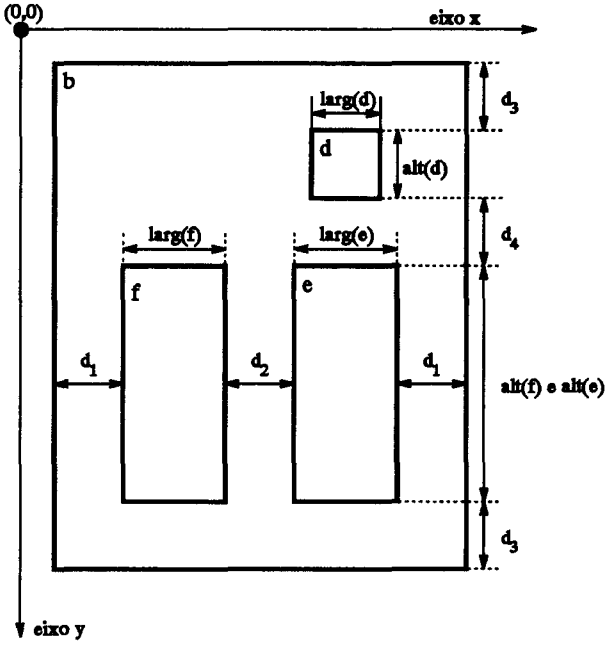



Figura 4.10: Representação de um leiaute métrico

Antes de apresentar o problema associado ao posicionamento de vértices é necessária a apresentação de alguns parâmetros envolvidos em tal problema (Figura 4.10), são eles:

- distância mínima horizontal entre vértices filhos e as bordas dos vértices pais ($d_1$);
- distância mínima horizontal entre vértices de mesmo nível ($d_2$);
- distância vertical entre vértices filhos e as bordas dos vértices pais ($d_3$);
- distância vertical entre vértices de mesmo nível ($d_4$);
- altura dos vértices folhas;
- largura dos vértices folhas.

Este problema pode ser ainda subdividido em dois subproblemas independentes:

- **Subproblema vertical.** Responsável pelo cálculo da posição vertical e do tamanho dos vértices;
- **Subproblema horizontal.** Responsável pelo cálculo da posição horizontal e da largura dos vértices.

---

```
procedimento LeiauteMetricoVertical(lVertices)
inicio
  nivel_zero = VERDADEIRO;
  nivel = 0;
  max = 0
  /*numero maior de nivel dos filhos em D*/
  num_nivel = ObtemNumeroNiveis(D);
  CriaListaVazia(lVertices2);
  para i de 1 a num_nivel faca
    nivel = nivel + 1;
    para cada v em lVertices faca
      se ''v tem filhos de nivel i'' entao
      ''Acrescenta em lVertices2, filhos de v que possuam nivel i''
    se nivel_zero == VERDADEIRO entao
      nivel_zero = FALSO;
      se Vazio(L) entao
        nivel = nivel - 1;
    para cada v em L faca
      PosicaoVertical(v) = BORDA_VERTICAL + max + (MEIO_VERTICAL * nivel);

    max = max + LeiauteMetricoVertical(lVertices2);
```

```
  altura_max = 0;
  para cada v em lVertices faca
    CalculaAltura(v);
    se Altura(v) > altura_max entao
      altura_max = Altura(v);
  para cada v em lVertices faca
    se TipoVertice(v) == FALSO entao
      Altura(v) = altura_max;
  retorna altura_max;
fim
```

---

A resolução do subproblema vertical é mais fácil do que o do subproblema horizontal, já que não tem relação com nenhuma característica estética. Um possível método para sua resolução pode ser visto no código apresentado através do procedimento **LeiauteMetrico-Vertical**. O método, para cada vértice não folha da hierarquia, baseia-se na determinação da posição relativa de seus vértices filhos, tendo como ponto de referência a sua posição. Uma vez tendo calculada toda as posições de seus vértices filhos, pode-se então definir a sua altura através da soma das alturas dos vértices com maior altura em cada nível da hierarquia local, formada pelos seus vértices filhos, acrescentando a distância vertical entre cada nível ($d_4$) e a distância entre a borda do vértice pai e os vértices filhos ($d_3$). O método requer que a altura, bem como a largura, de todos os vértices folhas, devam estar definidos previamente. Este método é recursivo, começando de baixo para cima na hierarquia formada pelo dígrafo de inclusão.

---

```
procedimento LeiauteMetricoHorizontal(v)
inicio
  CriaListaVazia(lVertices);
  para cada w em Filhos(v) faca
    se nao ListaVazia(Filhos(w)) entao
      Insere(w,lVertices);
    senao
      CalculaLargura(w);
  para cada w em lVertices faca
    LeiauteMetricoHorizontal(w);
  MetodoLeiautePrioritario(v);
fim

procedimento MetodoLeiautePrioritario(v)
```

```
inicio
  num_nivel = ObtemNumeroNiveis(v);
  num_heuristico = ''numero entre 1 e num_nivel'';
  para cada N_i de v faca
    InicializaMetodoLP(N_i)
  para i de 1 a num_nivel faca
    LeiauteLocal(N_i,BAIXO);
  para i de num_nivel a 1 faca
    LeiauteLocal(N_i,CIMA);
  para i de 1 a num_heuristico faca
    LeiauteLocal(N_i,BAIXO);

fim

procedimento LeiauteLocal(Nivel,direcao)
inicio
  enquanto nao ListaVazia(Nivel) faca
    v = ObtemVerticeMaiorPrioridade(Nivel);
    se direcao == BAIXO entao
      baricentro = BaricentroGlobalBaixo(v);
    senao
      baricentro = BaricentroGlobalCima(v);
      encontrado = FALSO;
    se baricentro < PosicaoHorizAbsoluta(v) entao
      pos = Posicao(v,Nivel) - 1;
      enquanto i > 0 e nao encontrado faca
        u = Nivel_pos;
        se Prioridade(u) ≥ Prioridade(v) entao
          encontrado = VERDADEIRO;
        senao
          pos = pos - 1;
      se encontrado e baricentro - PosicaoHorizAbsoluta(u) <
Distancia(u,v) entao
        para i de pos + 1 a Posicao(v,Nivel) faca
          w = Nivel_i;
          PosicaoHorizAbsoluta(w) = PosicaoHorizAbsoluta(u) +
Distancia(u,w);
      senao
```

```
        i = PosicaoHorizAbsoluta(v);
        para = FALSO;
        enquanto i > 1 e nao para entao
          w = Nivel_i;
          se baricentro - PosicaoHorizAbsoluta(w) < Distancia(w,v)
            PosicaoHorizAbsoluta(w) = baricentro - Distancia(w,v);
          senao
            para = VERDADEIRO;
    se baricentro > PosicaoHorizAbsoluta(v) entao
      pos = Posicao(v,Nivel) + 1;
      enquanto i > 0 e nao encontrado faca
        u = Nivel_pos;
        se Prioridade(u) ≥ Prioridade(v) entao
          encontrado = VERDADEIRO;
        senao
          pos = pos + 1;
      se encontrado e PosicaoHorizAbsoluta(u) - baricentro <
Distancia(v,u) entao
        para i de pos - 1 a Posicao(v,Nivel) faca
          w = Nivel_i;
          PosicaoHorizAbsoluta(w) = PosicaoHorizAbsoluta(u) -
Distancia(w,u);
      senao
        i = PosicaoHorizAbsoluta(v);
        para = FALSO;
        enquanto i ≤ Card(Nivel) e nao para entao
          w = Ni(i);
          se PosicaoHorizAbsoluta(w) - baricentro < Distancia(v,w)
            PosicaoHorizAbsoluta(w) = baricentro + Distancia(v,w);
          senao
            para = VERDADEIRO;
fim
```

Já o subproblema horizontal é bem mais complexo do que o vertical, pois envolve as características estéticas: Proximidade, Linha Reta e Balanceamento. Há dois métodos para solucionar esse problema [53]: método *Programação Quadrática*[1] e método *Leiaute Prioritário*[2]. Dado que o desempenho do primeiro é bem inferior ao segundo, que é

[1] Do inglês: *Quadratic Programming Method*
[2] Do inglês: *Priority Layout Method*

baseado em heurísticas, optou-se pela aplicação do segundo método.

O método *Leiaute Prioritário* é baseado no cálculo do baricentro dos vértices, semelhante ao empregado durante a fase de ordenação. Contudo, ao invés de se levar em consideração a posição relativa do vértice na ordem de cada nível da hierarquia local, leva-se em consideração a sua posição absoluta no leiaute. O trecho de código que implementa este método pode ser visto através do procedimento MetodoLeiautePrioritario.

Para o cálculo dos valores do baricentro cima e baixo de cada vértice, utiliza-se as funções Baricentro Global Cima ($BG_c$) e Baricentro Global Baixo ($BG_b$), respectivamente, e que são definidas como:

$$BG_c(v) = \begin{cases} \frac{\sum_{(u,v)\in Fc(v)} \chi(u)}{|Fc(v)|} & \text{se } |Fc(v)| > 0 \\ 0 & \text{caso contrário} \end{cases} \tag{4.11}$$

$$BG_b(v) = \begin{cases} \frac{\sum_{(v,u)\in Fd(v)} \chi(u)}{|Fd(v)|} & \text{se } |Fd(v)| > 0 \\ 0 & \text{caso contrário} \end{cases} \tag{4.12}$$

Onde, $v \in V$, $Fc(v) \subset F^d$ é o conjunto de arestas convergentes a $v$, $Fd(v) \subset F^d$ é o conjunto de arestas divergentes a $v$ e $\chi(v)$ é a posição global de $v$ no eixo horizontal (eixo x) do leiaute.

A idéia fundamental do algoritmo é a aplicação seqüencial do método a cada nível da hierarquia local. A cada aplicação do método, a escolha do vértice é determinada pelo vértice com maior prioridade. De modo a otimizar a característica estética Linha Reta, deve-se sempre dar maior prioridade ao posicionamento dos vértices falsos. Assim, tende-se a diminuir o número de "quebras" nas linhas. Aos demais vértices, suas prioridades são associadas ao grau que cada um possui.

O método começa nos vértices cujo os vértices filhos são vértices folhas e, termina no vértice raiz, ou seja, a aplicação começa de baixo para cima na hierarquia formada pelas arestas de inclusão.

```
procedimento LeiauteGlobal(v)
inicio
  para cada w em Filhos(v) faca
    PosicaoHorizAbsoluta(w) = PosicaoHorizAbsoluta(w) +
PosicaoHorizAbsoluta(v);
    PosicaoVertical(w) = PosicaoVertical(w) + PosicaoVertical(v);
    LeiauteGlobal(w);
fim
```

Por fim, um ajuste em todo leiaute é necessário, já que as posições calculadas para cada vértice do dígrafo, através dos dois métodos acima apresentados, tem como único ponto de referência a posição de seu vértice pai. Assim, há a necessidade de um reescalonamento de modo a adequar todos os vértices do dígrafo a um único posicionamento global. Este reescalonamento basea-se em deslocar a posição dos vértices filhos em relação a posição de seu vértice pai, começando no vértice raiz e descendo na hierarquia do dígrafo de inclusão até os vértices folhas. O código relativo a este processo pode ser visto através do procedimento LeiauteGlobal.

### 4.2.3 Mapeamento e Verificação entre as Estruturas

Dado que o algoritmo de Sugiyama foi desenvolvido visando a uma estrutura baseada em dígrafos, há a necessidade de mapear as estruturas tanto do modelo estrutural como a do modelo de controle para uma correspondente estrutura de dígrafos. Este processo deve ser realizado de maneira que seja possível a recuperação da informação original, ou seja, transformar a estrutura de dígrafos nas suas correspondentes estruturas de modelo estrutural e de modelo de controle.

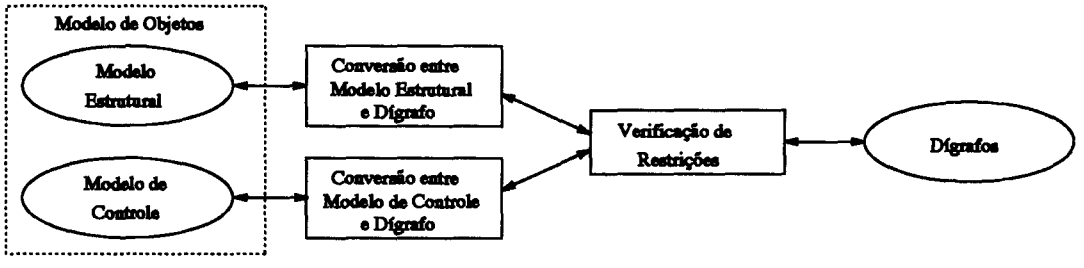



Figura 4.11: Mapeamento entre as estruturas de dígrafo e dos modelos estrutural e de controle

Infelizmente, este mapeamento não é obtido de forma direta, devido às incompatibilidades existentes entre as estruturas. Por isso, há a necessidade de procedimentos de conversão que controlem o mapeamento entre as estruturas, de modo que nenhuma informação seja perdida ou indevidamente adulterada durante o processo (Figura 4.11). Dado que também existem diferenças entre as estruturas de cada um dos modelos, há necessidade de diferentes procedimentos de conversão para cada uma das estruturas. Finalmente, deve-se verificar se todas as restrições impostas pelo algoritmo de Sugiyama estão sendo devidamente obedecidas.

### Mapeamento entre Modelo Estrutural e Dígrafo

O mapeamento entre a estrutura do modelo estrutural e a estrutura de dígrafo é feito da seguinte maneira: cada classe corresponde a um vértice e cada relacionamento corresponde a uma aresta de adjacência. Não existe aresta de inclusão durante o mapeamento.

Durante o mapeamento, podem surgir certos problemas. Um deles se deve a não existência de orientação nos relacionamentos de associação. Assim, deve-se definir qual será a orientação da correspondente aresta de adjacência para cada um destes relacionamentos. Contudo, este problema é um pouco mais complexo do que aparenta ser. A determinação da orientação de uma única aresta pode mudar o resultado (leiaute) gerado pelo algoritmo. Assim, as várias combinações das orientações de todas as arestas que representam relações de associações podem gerar um conjunto de diferentes resultados. Dentre este conjunto, alguns certamente vão possuir leiautes considerados melhores do que os demais. Assim, seria interessante obter uma combinação de orientações de arestas que gere o resultado de melhor leiaute. Contudo, não é possível determinar uma regra exata que indique essa combinação. Tentativas foram feitas para se criar procedimentos heurísticos para a sua determinação, sem, no entanto, alcançar nenhum resultado prático considerado "satisfatório". Optou-se, portanto, por gerar essa combinação de forma aleatória.

Outro problema é devido às relações de generalização/especialização e agregação. Optou-se por definir a orientação de suas correspondentes arestas como sendo dos vértices correspondentes às classes bases para os vértices correspondentes às classes derivadas e dos vértices correspondentes às classes agregadas para os vértices correspondentes às classes componentes, respectivamente. Com isto, o algoritmo gera a hierarquia, fortemente tendenciado a posicionar as classes bases e agregadas nos níveis superiores e as classes derivadas e componentes nos níveis inferiores da mesma. Essa solução foi escolhida por ser esta a forma mais comumente utilizada nas representações para tais tipos de relações.

### Mapeamento entre Modelo de Controle e Dígrafo

O mapeamento entre a estrutura do modelo de controle e a estrutura de dígrafo é bem mais simples do que o mapeamento acima, já que as estruturas podem ser mapeadas quase que diretamente. Neste caso, cada estado é mapeado para um vértice e cada transição é mapeada para uma aresta de adjacência, onde a orientação da aresta corresponde exatamente à orientação da transição. Por fim, cada relação de generalização/especialização ou de agregação é mapeada para uma aresta de inclusão, cuja orientação é definida como sendo do vértice correspondente ao superestado para o vértice correspondente ao subestado.

### Verificação das Restrições

Uma vez feita o mapeamento das estruturas dos modelos estrutural e de controle, há ainda a necessidade de se verificar se as restrições impostas ao dígrafo, pelo algoritmo de Sugiyama, estão sendo adequadamente obedecidas. Caso isto não ocorra, há a necessidade da execução de certos procedimentos, para que o dígrafo obedeça às restrições. Contudo, é importante ressaltar que toda a operação de alteração na estrutura do dígrafo deve ser armazenada de alguma forma, de modo que se possa refazer a estrutura original do mesmo.

As restrições e os respectivos procedimentos adotados são apresentados a seguir:

- **Dígrafo de inclusão deve ser uma árvore com apenas um vértice raiz.** Um falso vértice raiz deve ser incluído no dígrafo e falsas arestas de inclusão devem ligar este vértice raiz aos vértices que pertencem ao primeiro nível da árvore de inclusão. Durante o processo de retorno, tanto o vértice falso raiz quanto as arestas de inclusão falsas devem ser removidas do dígrafo.

- **Não deve existir nenhuma relação de adjacência entre vértices ancestrais e descendentes.** Toda aresta que representa esse tipo de relação deve ser removida do dígrafo e armazenada adequadamente. Contudo, durante a operação de recuperação da estrutura original do dígrafo, a informação de rota de cada uma destas arestas deve ser definida como sendo um único segmento de reta ligando os seus respectivos vértices.

- **Não deve existir arestas de adjacência múltiplas.** Quando houver mais de uma aresta ligando pares de vértices, deve-se substituí-las por uma única aresta representando todas as demais. Durante o processo de retorno, essa aresta deve ser então substituída por todas correspondentes arestas originais, com a mesma informação de rota.

## 4.2.4 Implementação

Toda a modelagem do algoritmo foi realizada utilizando-se a metodologia orientada a objetos OMT [50]. Para a codificação do componente Gerador de Leiaute foi utilizada a linguagem C++ [20].

## 4.2.5 Consideração Final

Alguns resultados gerados pela implementação do algoritmo realizada neste trabalho diferem dos resultados apresentados na especificação do mesmo [53]. Por exemplo, para

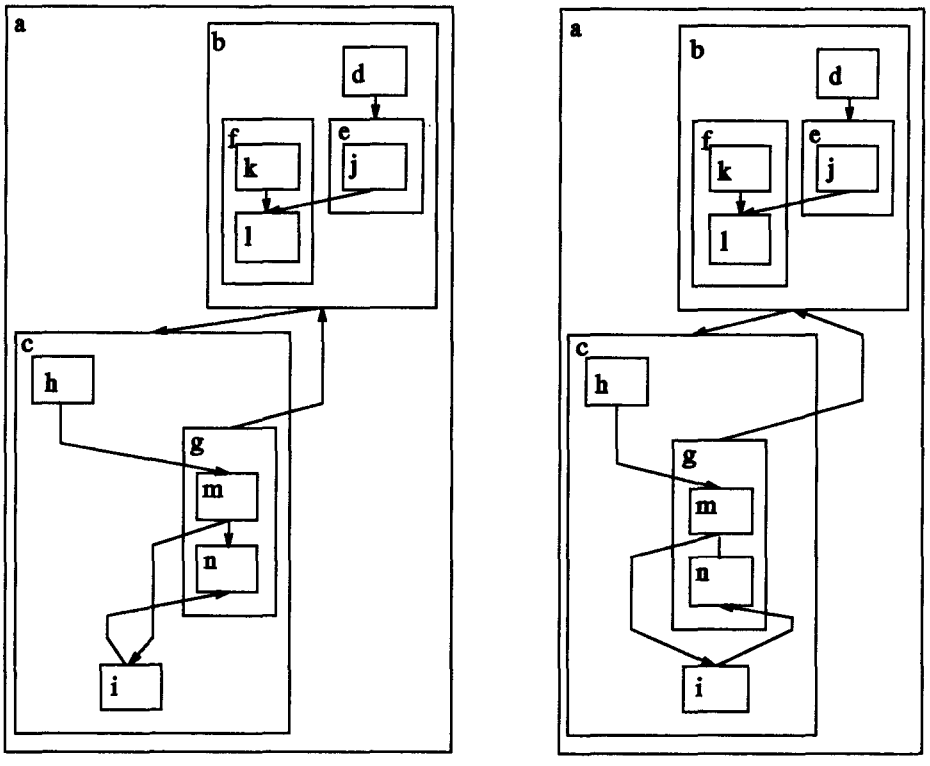



Figura 4.12: Diferença entre dos resultados gerados pela implementação e especificação do algoritmo

o dígrafo composto apresentado na Figura 1.12, a implementação gerou o leiaute representado pela Figura 4.12a, enquanto que a especificação mostra como resultado final do algoritmo o leiaute representado pela Figura 4.12b. Como pode-se perceber, há duas diferenças principais entre os dois leiautes gerados:

1. O posicionamento do vértice $g$
2. O posicionamento do vértice $i$

Em conseqüência destas duas diferenças foram gerados roteamentos diferentes para as arestas (divergentes e convergentes) dos vértices g e i, tais como (g,b), (i,n), bem como para as arestas de seus vértices descendentes no dígrafo de inclusão.

Durante a fase de ordenação, mais especificamente durante a execução do método *Splitting*, os vértices com $\Delta > 0$ e $\Delta < 0$ devem ser posicionados mais a direita e mais a esquerda, respectivamente, na ordem ($\sigma$). Calculando-se o valor de $\Delta$ para os vértices $g$, $i$ e $h$, tem-se: $\Delta(g) = 1$, $\Delta(i) = 0$ e $\Delta(h) = 0$. Com isto, e a partir das propriedades 4.1 e 4.8, tem-se que o vértice g necessariamente tem de ser colocado na posição mais a direita dentro da sua ordem $\sigma$, o que ocorre no leiaute da Figura 4.12a. Já na Figura 4.12b, o

vértice posicionado mais a esquerda, trata-se de um vértice falso. Isto pode ser notado através do roteamento da aresta $(i, n)$. Tal vértice possui $\Delta = 0$, portanto não pode ser posicionado mais a esquerda de g. Assim, pode-se deduzir que o leiaute da Figura 4.12b não obedece as regras impostas pelo algoritmo de Sugiyama. O posicionamento diferente do vértice $i$ é gerado em conseqüência ao diferente posicionamento diferente do vértice $g$.

## 4.3 Gerador de Código

A função do componente Gerador de Código é gerar os metaprogramas a partir das informações armazenadas na estrutura de dados neutra. Para cada modelo de objetos são gerados dois metaprogramas, estrutural e de controle, que armazenam as informações relativas aos modelos estrutural e de controle, respectivamente. Os metaprogramas são responsáveis por armazenar, em forma de esquema, as informações do modelo de objetos no ambiente de programação Stabilis/Vigil [4], através das bibliotecas fornecidas pelo ambiente. Além disso, ele é responsável também pela geração do programa cuja função é invocar os métodos geradores de código do ambiente Stabilis/Vigil.

Devido ao atual desenvolvimento de trabalhos visando modificações na estrutura do ambiente de programação Stabilis/Vigil, a linguagem de descrição dos metaprogramas ainda não foi totalmente definida. Apesar disto, é apresentado a seguir, um esboço de parte da estrutura básica do código gerado para cada um destes programas, tendo em vista dar uma possível visão geral do mesmo. Contudo, deve estar claro que o modelo de geração de código apresentado pode não estar em perfeita conformidade com a interface de programação do ambiente, estando sujeito a futuras alterações.

As informações apresentadas na forma sublinhada devem ser substituídas de acordo com as informações contidas em cada uma das instâncias das classes da estrutura de dados neutra.

### 4.3.1 Metaprograma Estrutural

Para cada modelo de objetos é gerado o seguinte código:

```
mdl_nome = new Model("Model(name == 'nome')", view, BIRTH, oph);
```

Onde nome é substituído pelo nome do modelo de objetos. Em seguida, é apresentado o código gerado para cada classe, atributo, método e relacionamento do modelo estrutural, ou seja, para cada uma das instâncias das classes Class, Attribute, Method, Association, LooseAggregation, TightAggregation, e Specialization.

### Class

```
Class* cl_nome = new Class("Class(name = 'nome')", view, BIRTH, oph);
```

Onde nome é substituído pelo nome da classe.

### Attribute

```
TipoAttribute* at_nome = new TipoAttribute("TipoAttribute(name = 'nome')",
view, BIRTH, oph);
oph += cl_nomeclasse→relate(Ättribute", at_nome);
```

Onde nome é substituído pelo nome do atributo, nomeclasse é substituído pelo nome da classe a qual o atributo pertence e Tipo pelo seu tipo, por exemplo: Interger, no caso de inteiro, e String, no caso de cadeia de caracteres.

### Method

```
Method* mt_id = new Method("Method(name = 'assinatura'","tiporetorno",
view, BIRTH, oph);
oph += cl_nomeclasse→relate("Method",at_nome);
```

Onde id é um identificador único gerado automaticamente para cada método no modelo de objetos, assinatura é a assinatura (protótipo) da função, sem o tipo de retorno, e tiporetorno é tipo de retorno do método.

### Association

```
Association rel_id = new Association (Ässociation(left_role ==
'papel_esq' && left_min_card == card_min_esq && left_max_card ==
card_max_esq && right_role == 'papel_dir' && right_min_card ==
card_min_dir && right_max_card == card_max_dir && key == 1)",
view, BIRTH, oph);
oph += rel_id→relate("LeftClass", ClasseDireita);
oph += rel_id→relate("RightClass", ClasseEsquerda);
```

Onde id é substituído por um identificador gerado automaticamente para cada relacionamento no modelo de objetos, papel_esq e papel_dir indicam os papeis das classes no relacionamento, card_min_esq e card_min_dir descrevem as cardinalidades mínimas, card_max_esq e card_max_dir descrevem as cardinalidades máximas e ClasseDireita e ClasseEsquerda representam as classes participantes do relacionamento de associação.

**LooseAggregation**

```
LooseAggregation rel_id = new LooseAggregation(,,,, view, BIRTH, oph);
oph += rel_id→relate("LeftClass", ClasseDireita);
oph += rel_id→relate("RightClass", ClasseEsquerda);
```

Onde id é substituído por um identificador gerado automaticamente para cada relacionamento no modelo de objetos e ClasseDireita e ClasseEsquerda representam as classes participantes do relacionamento de associação.

**TightAggregation**

```
TightAggregation rel_id = new TightAggregation(,,,, view, BIRTH, oph);
oph += rel_id→relate("LeftClass", ClasseDireita);
oph += rel_id→relate("RightClass", ClasseEsquerda);
```

Onde id é substituído por um identificador gerado automaticamente para cada relacionamento no modelo de objetos e ClasseDireita e ClasseEsquerda representam as classes participantes do relacionamento de associação.

**Specialization**

```
Classification rel_id = new Classification(view, BIRTH, oph);
oph += rel_id→relate("LeftClass", ClasseDireita);
oph += rel_id→relate("RightClass", ClasseEsquerda);
```

Onde id é substituído por um identificador gerado automaticamente para cada relacionamento no modelo de objetos e ClasseDireita e ClasseEsquerda representam as classes participantes do relacionamento de associação.

### 4.3.2 Metraprograma de Controle

Para cada modelo de controle o seguinte código é gerado:

```
TransitionSystem* cntr_nome = new TransitionSystem("TransitionSystem(
name = 'nome')", BIRTH, oph);
Class* cl_nome = new Class("Class(name = 'nome')", REINCARNATE, view,
oph);
oph += cl_nome→relate("TransitionSystem", cntr_nome);
```

Onde nome é substituído pelo nome da classe a qual o modelo de controle está associado.

Em seguida, é apresentado o código gerado para cada estado e transição do modelo de controle, ou seja, para cada uma das instâncias das classes State e Transition.

**State**

```
State* st_id = new State("State(name = 'nome')", view, BIRTH, oph);
```

Onde id é substituído por um identificador gerado automaticamente para cada estado. Caso o estado seja um subestado de outro estado, pode ser gerada um entre dois tipos de código, dependendo do tipo de relação existente entre os estados. Se a relação for de especialização/generalização, é gerado o seguinte código:

```
*oph += st_id_superestado→relate("SubState", st_id);
```

Já se a relação for de agregação, então o seguinte código é gerado:

```
*oph += st_id_superestado→relate("StateAggregatee", st_id);
```

Onde id_superestado é substituído pelo identificador do seu superestado.

**Transition**

```
GuardedAction tr_id = new GuardedAction("GuardedAction(guard =
'condição' && action = 'ação')", view, PROVIDE, oph);
oph += tr_id→relate("Predecessor", st_id_predecessor);
oph += tr_id→relate("Successor", st_id_sucessor);
```

Onde id é substituído por um identificador gerado automaticamente, condição representa a condição associada à transição, ação representa a ação associada à transição e id_predecessor e id_sucessor representam os identificadores dos estados predecessor e sucessor, respectivamente.

### 4.3.3 Programa para Invocação de Geração de Código

Este programa, gerado pelo componente Gerador de Código, é responsável por invocar a rotina do ambiente de programação necessária para a geração de código relativo ao programa distribuído. O ambiente de programação faz uso dos esquemas (estrutural e de controle) para fazer a devida geração de código.

A seguir é apresentado parte da estrutura básica do programa gerado pelo componente:

```
main (int argc, char* argv[]) {
View* view = 0;
OpHistory oph;
String path;
```

```
if (argc == 1) path = ".";
else path = argv[1];
Model* mdl_nome = new Model("Model(name == 'nome')",view,
REINCARNATION, &oph);
oph += mdl_nome->gen_code(path, 1);
return 0;
}
```

Onde nome é substituído pelo nome do modelo.

## 4.4 Gerenciador de Persistência

O componente Gerenciador de Persistência é responsável pelo armazenamento das informações dos modelos de objetos em memória estável, bem como pela sua recuperação. Essa operação é executada com o auxílio de funcionalidades fornecidas pelo módulo **Object Store** de Arjuna (ver Seção 2.1.1). Todos os objetos do modelo de objetos são armazenados em um repositório de objetos fornecido pelo módulo **Object Store**.

Outra funcionalidade fornecida por este componente, é um mecanismo bastante simples de controle de versões dos modelos de objetos armazenados no repositório de objetos. Essa funcionalidade é implementada utilizando a classe **VersionModel** (Figura 4.1). Existe uma única instância desta classe para cada modelo de objetos. Esta classe possui um atributo que indica a última versão do referido modelo. As várias versões de um mesmo modelo são identificadas através de um atributo, que indica o número da versão, existente na classe **Model**.

Através deste mecanismo, o programador pode salvar no repositório de objetos quantas versões do modelo de objetos ele queira. Para fazer a recuperação de uma determinada versão de um modelo, basta utilizar como chave de busca no repositório a composição do nome e versão do modelo.

## 4.5 Resumo

Neste capítulo foi apresentado, de forma detalhada, cada um dos componentes que compõe o módulo Aplicação do editor gráfico. Primeiramente, foi apresentada a estrutura de dados que compõe o componente Estrutura de Dados Neutra e qual é o objetivo desta estrutura. Em seguida, foi apresentado o algoritmo de Sugiyama, que foi o algoritmo escolhido para implementar a função de traçado de leiaute automático do editor gráfico. Além disso, foram apresentados alguns questionamentos sobre as diferenças observadas entre os resultados apresentados na sua especificação e os obtidos através da sua implementação.

Por fim, foram apresentadas as estruturas internas dos componentes Gerador de código e Gerenciador de Persistência.

No Capítulo seguinte é apresentada a implementação de um problema utilizando o editor gráfico, em vista a mostrar as facilidades obtidas com a sua utilização.

# Capítulo 5

# Exemplo

Este capítulo tem por objetivo mostrar como a utilização do editor pode facilitar o processo de desenvolvimento de programas distribuídos orientados a objetos no ambiente Stabilis/Vigil. Optou-se pela apresentação de um problema real e bastante conhecido: o problema do Jantar dos Filósofos.

Primeiramente é feita uma breve descrição do problema e, em seguida, a apresentação do código dos metaprogramas que representam o modelo de objetos utilizado na solução do problema. Por fim, é apresentado como a implementação desta solução poderia ser realizada através do uso do editor gráfico.

## 5.1 Jantar dos Filósofos

O problema do Jantar dos Filósofos [17] é um problema clássico de programação concorrente. Embora a princípio possa ser considerado um mero entretenimento, ele fornece meios de comparação com vários formalismos relativa a programação concorrente. Isto valida a sua utilização como um exemplo real, através do qual pode-se mostrar a utilidade do editor gráfico na construção de programas distribuídos orientados a objetos.

### 5.1.1 Descrição do Problema

Esse problema envolve a alocação coordenada de recursos compartilhados e escassos (garfos) entre cinco filósofos arranjados em um círculo. Em um determinado momento um filósofo pode estar pensando, faminto ou comendo. Para passar de faminto para comendo, o filósofo deve possuir os garfos da direita e da esquerda. O fato dos filósofos compartilharem garfos e da obrigatoriedade de adquirir dois garfos antes que possam comer pode conduzir a conflitos entre eles, já que somente um filósofo pode ter a posse de um garfo em um dado momento. Dado que este é um problema clássico de exclusão mútua, não é

necessário tornar explícitas as condições que condicionam a sua correção.

### 5.1.2 Solução do Problema

A solução apresentada a seguir é uma dentre as várias soluções existentes para o problema dos Filósofos. Esta solução é baseada na descrição apresentada por Chandy e Misra [5].

#### Modelo Estrutural

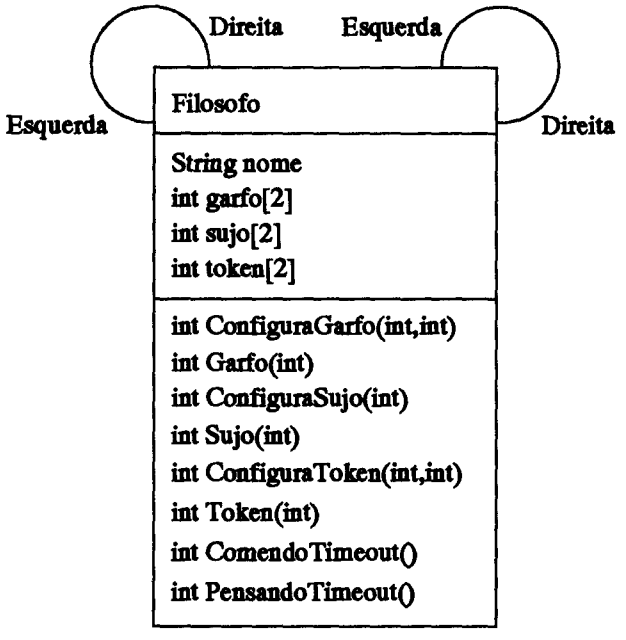



Figura 5.1: Modelo Estrutural para o Problema do Jantar dos Filósofos

Um modelo estrutural pode ser usado para representar o arranjo circular dos filósofos (Figura 5.1). Basicamente, o modelo de objetos construído consiste da classe Filósofo; instâncias dessa classe representam cada um dos 5 filósofos. Os dois relacionamentos de associação, Esquerda e Direita, modelam a vizinhança entre filósofos.

Nesta solução, a classe Filósofo possui os seguintes atributos:

- nome: identifica unicamente cada instância da classe;
- garfo: indica se o filósofo tem a posse ou não dos garfos que compartilha com seus vizinhos;
- sujo: indica se os garfos estão sujos ou não;
- token: indica que o filósofo tem o direito de pedir o garfo ao seu vizinho.

Para cada atributo onde há algum tipo de interesse de gerenciamento, existe um sensor e um atuador. Por exemplo, para o atributo **garfo**, existe o sensor Garfo(...) e o atuador ConfiguraGarfo(...). Os demais sensores e atuadores podem ser vistos na representação do modelo estrutural (Figura 5.1).

### Modelo de Controle

O modelo de controle para a solução do problema do Jantar dos Filósofos é baseado na seguinte descrição do comportamento do problema, seguindo Chandy e Misra [5]: "Um garfo está limpo ou sujo. Um garfo sendo utilizado para comer está sujo e continua sujo até ser limpo. Um garfo limpo permanece limpo até ser usado para comer. Um filósofo limpa o garfo quando o repassa para outro filósofo; ele é higiênico. Enquanto come, um filósofo não satisfaz solicitações de garfos.

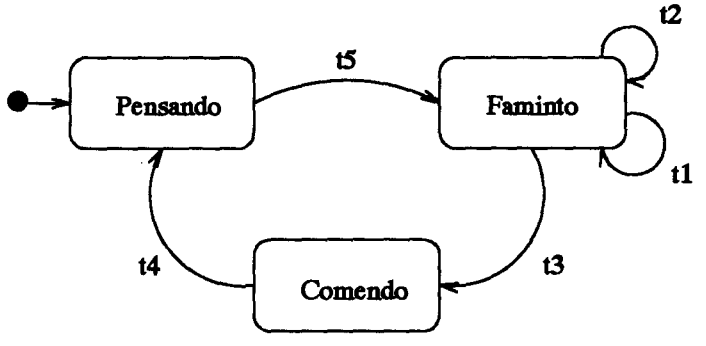



Figura 5.2: Modelo de Controle para o Problema do Jantar dos Filósofos

O modelo de controle que expressa a solução baseada na descrição acima pode ser visto na Figura 5.2. As transições deste modelo são expressas da seguinte maneira:

- **t1.** Requisitando um garfo ao vizinho do lado l:
  (Token(l) && !Garfo(l)) /
  {vizinho$_l$.ConfiguraToken(OutroLado(l),VERDADEIRO); ConfiguraToken(l,FALSO);}

- **t2.** Liberando um garfo do lado *l*:
  (Token(l) && Sujo(l)) /{ConfiguraSujo(l,FALSO); ConfiguraGarfo(l,FALSO);
  vizinho$_l$.ConfiguraGarfo(OutroLado(l),VERDADEIRO);}

- **t3.** Passando do estado faminto para comendo:
  (Garfo(ESQUERDA) && Garfo(DIREITA)) /
  {ConfiguraSujo(ESQUERDA,VERDADEIRO);
  ConfiguraSujo(DIREITA,VERDADEIRO);}

- **t4.** Passando do estado comendo para pensando:
  (ComendoTimeout()) / {}

- **t5.** Passando do estado pensando para faminto:
  (PensandoTimeout()) / {}

Onde l assume representa o lado do relacionamento com os filósofos vizinhos e assume um entre dois valores: ESQUERDA ou DIREITA.

### 5.1.3 Programando a Solução Apresentada

Após o término do projeto orientado a objetos, obtém-se um modelo de objetos que representa a solução encontrada para o problema. Programá-la utilizando o editor é bastante rápido e direto. Basta editar o modelo de objetos no mesmo (Figuras 5.1 e 5.2). Para isto, o programador deve selecionar a opção para desenho de classe no menu ou na barra de ícones *DrawBar* e, em seguida, clicar o *mouse* em algum ponto do área de trabalho do modelo estrutural. Com isto, uma janela de diálgo é apresenta onde se deve preencher as informações relativas à classe (Figura 5.3).

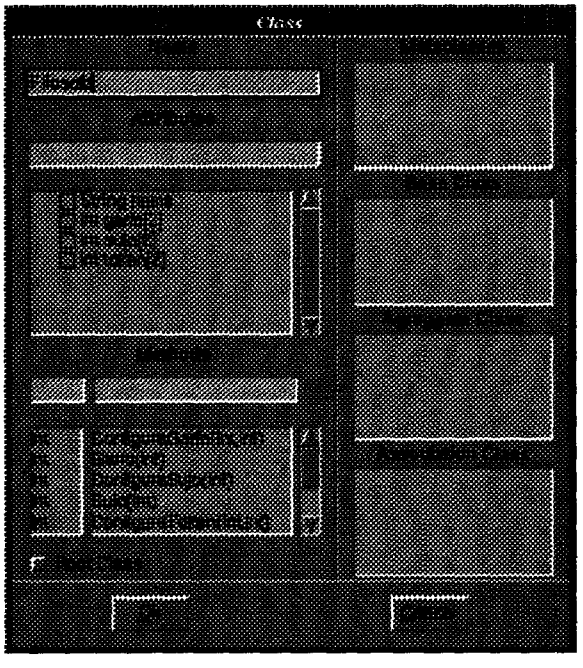


Figura 5.3: Janela de diálogo para especificação de classes

A seguir, o programador deve especificar os relacionamentos de associação que modelam a vizinhança entre os filósofos. Durante a criação de um relacionamento de associação,

uma janela de diálogo é apresentada para que sejam preenchidas as informações do relacionamento (Figura 5.4). A especificação de todo o modelo de estrutural no editor gráfico pode ser vista através da Figura 5.5.

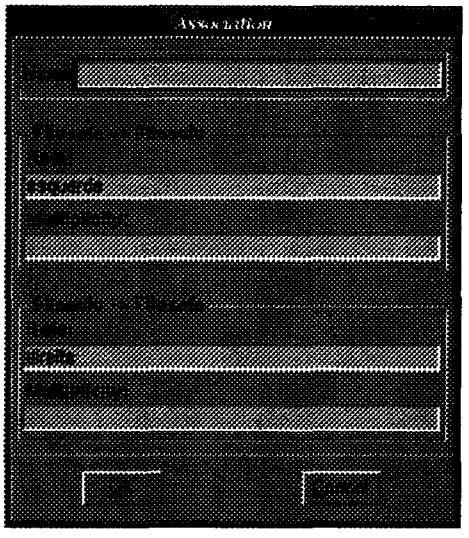


Figura 5.4: Janela de diálogo para especificação de relacionamentos de associação

Para fazer a especificação do modelo de controle da classe **Filosofo** o programador deve então selecioná-la e acessar a área de trabalho do modelo de controle. Para criar os estados **Pensando**, **Faminto** e **Comendo**, o programador deve selecionar a opção para desenho de estado no menu ou na barra de ícones *DrawBar* e clicar o *mouse* em alguma região da área de trabalho do modelo de controle. Uma janela de diálogo é então aberta para que se possa fazer as especificações das informações relativas ao estado (Figura 5.6).

Para a criação das transições, o programador deve selecionar a opção de desenho de transição através do barra de menu ou da barra de ícone *DrawBar*, clicar no estado de origem, arrastar o mouse até o estado destino e então soltar o botão do *mouse*. Uma janela de diálogo é então apresentada para que o programador coloque as informações da transição (Figura 5.7). Na Figura 5.8 pode ser vista a especificação de todo o modelo de controle no editor gráfico.

Uma vez que o modelo de objetos esteje devidamente especificado no editor gráfico, o programador deve então ativar a geração de código através da opção *Code Generation* da barra de menu. Parte dos códigos gerados dos metaprogramas podem ser vistos nos Programas 5.1 e 5.2. O trecho do programa de invocação de geração de código pode ser visto através do Programa 5.3. A não utilização do editor gráfico implica no programador em codificar manualmente os metaprogramas e o programa de invocação de geração de código.

Como pode-se perceber através do exemplo acima, a especificação de um simples mo-

```
main () {
  /* declaracao de variaveis locais foi omitida */
  Model* mdl_jantarfilosofos = new ("Model(name =
'JantarFilosofos')",...);
  Class* cl_filosofo = new Class("Class(name = 'Filosofo')",...);
  StringAttribute* at_name = new StringAttribute("
    StringAttribute(name = 'name' & key = 1)",...);
  *oph += cl_filosofo→relate("Attribute",at_name);
  /* declaracao dos demais atributos da classe filosofo foi omitida */
  /* declaracao dos metodos da classe filosofo foi omitida */
  Association* rel_1 = new Association("
    Association(left_role = 'esquerda' && left_min_card = 1 &&
      left_max_card = 1 && right_role = 'direita' && right_min_card = 1
      && right_max_card = 1 && key == 1
    )",cl_filosofo,cl_filosofo,...);
  Association* rel_2 = new Association("
    Association(left_role = 'direita' && left_min_card = 1 &&
      left_max_card = 1 && right_role = 'esquerda' && right_min_card = 1
      && right_max_card = 1 && key == 1
    )",cl_filosofo,cl_filosofo,...);
}
```

Programa 5.1: Metaprograma Estrutural para o Jantar dos Filósofos

```
main() {
  /* declaracao de variaveis locais foi omitida */
  Model* mdl_jantarfilosofos = new ("Model(name =
'JantarFilosofos')",...);
  Class* cl_filosofo = new Class("Class(name = 'Filosofo')",...);
  *oph += mdl_jantarfilosofos→relate("RootClass",cl_filosofo);
  TransitionSystem* cntr_filosofo = new TransitionSystem("
    TransitionSystem(name = 'Filosofo')",...);
  *oph += cl_filosofo→relate("TransitionSystem",cntr_filosofo);
  State* st_pensando = new State("State(name = 'Pensando')",...);
  State* st_faminto = new State("State(name = 'Faminto')",...);
  State* st_comendo = new State("State(name = 'Comendo')",...);
  GuardedAction* tr_t3 = new GuardedAction("
    GuardedAction(guard = 'Garfo(ESQUERDA) && Garfo(DIREITA)' &&
    action = '{ConfiguraSujo(ESQUERDA,VERDADEIRO);
    ConfiguraSujo(DIREITA,VERDADEIRO');}')",...);
  *oph += tr_t3→relate("Predecessor",st_faminto);
  *oph += tr_t3→relate("Successor",st_comendo);
  /* declaracao das demais transicoes foi omitida */
}
```

Programa 5.2: Metaprograma de Controle para o Jantar dos Filósofos

```
main (int argc, char* argv[]) {
  View* view = 0;
  OpHistory oph;
  String path;

  if (argc == 1) path = ".";
  else path = argv[1];
  Model* mdl_filosofo = new Model("Model(name == 'filosofo')", view,
    REINCARNATION, &oph);
  oph += mdl_filosofo→gen_code(path, 1);
  return 0;
}
```

Programa 5.3: Programa de invocação de geração de código para o problema dos filósofos

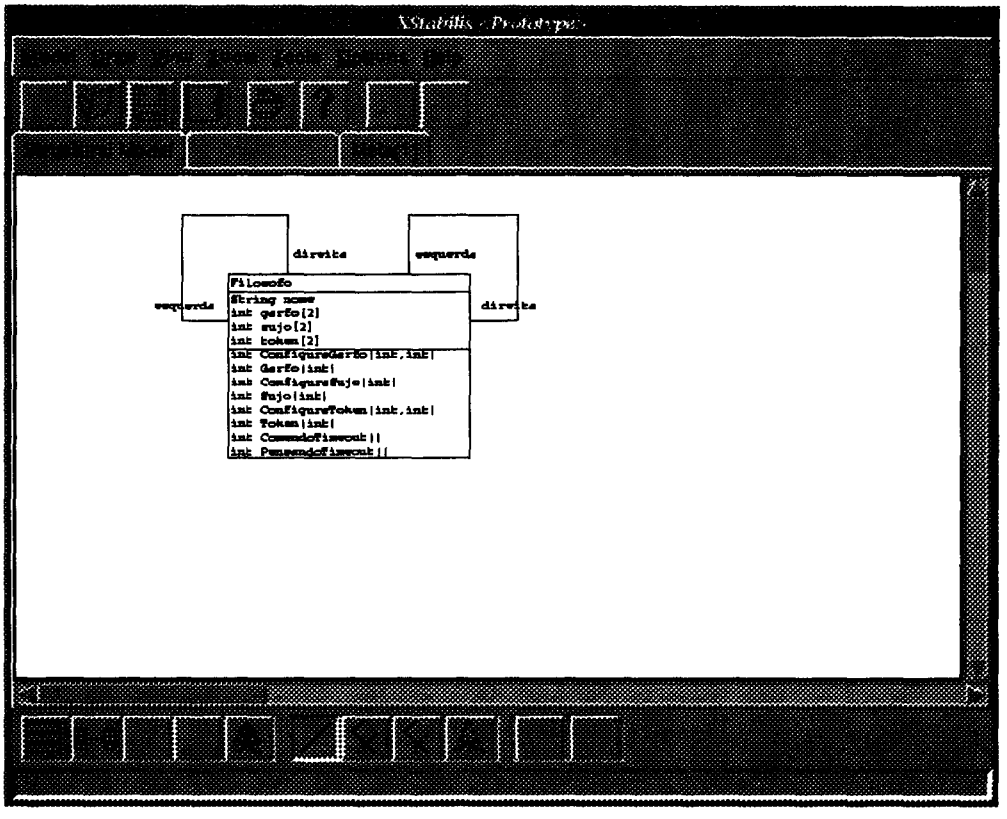



Figura 5.5: Visualização da especificação do modelo estrutural da classe Filosofo

delo de objetos no ambiente de programação, sem a utilização do editor, implica na construção de metaprogramas complexos e de difícil manutenção. Além disso, a visualização do modelo de objetos a partir do metaprograma é bem mais difícil que a interpretação dos metaprogramas tomando como referência a sua representação visual (diagrama).

## 5.2 Resumo

Neste capítulo foi apresentado o desenvolvimento de um exemplo, o Jantar dos Filósofos, com o intuito de exemplificar e validar a utilização do editor. Além disso, foi apresentado parte do conteúdo dos metaprogramas relativo ao modelo de objetos que implementa a solução adotada para o problema do Jantar dos Filósofos. Através deste exemplo, torna-se claro as vantagens da utilização do editor e o quão é mais difícil a programação no ambiente Stabilis/Vigil sem a sua utilização.

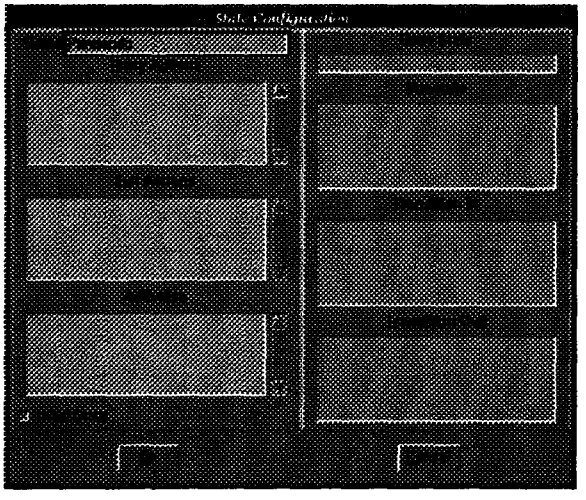


Figura 5.6: Janela de diálogo para especificação de estados

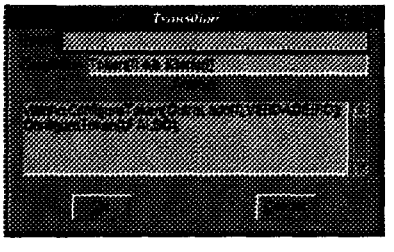


Figura 5.7: Janela de diálogo para especificação de transições

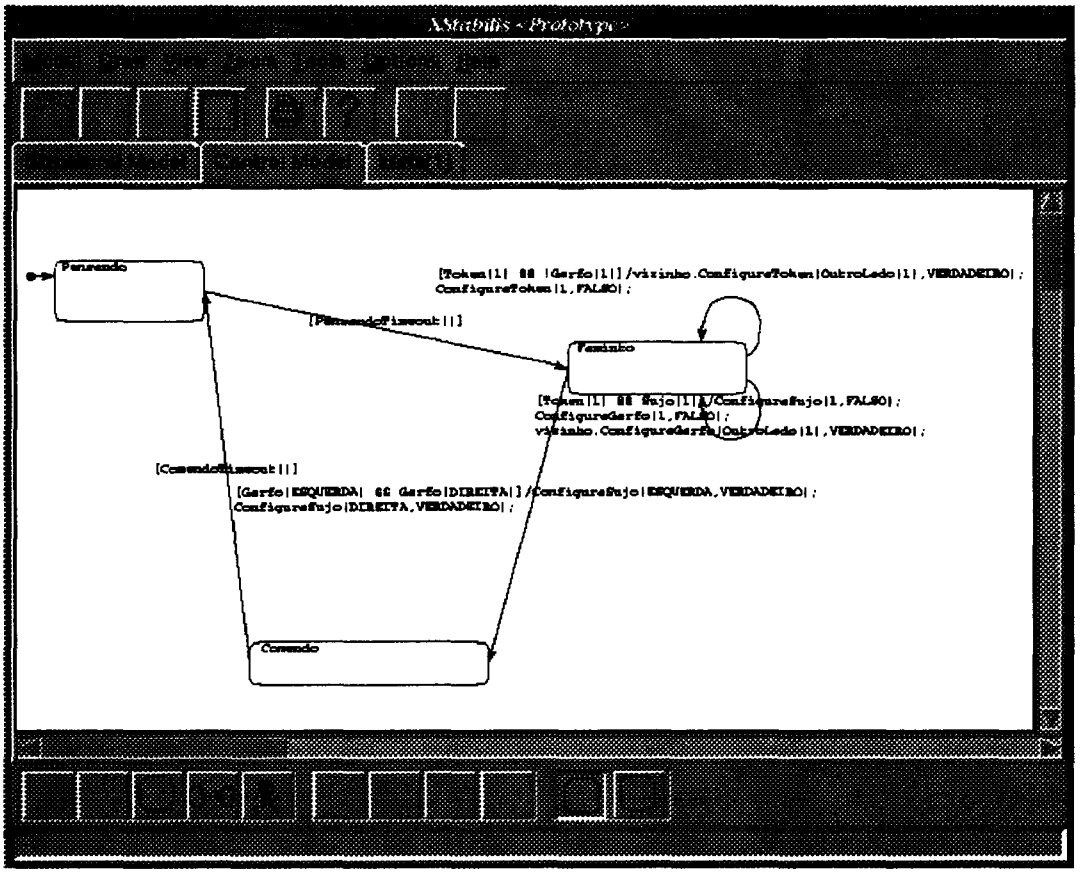



Figura 5.8: Visualização da especificação do modelo de controle da classe Filosofo

# Capítulo 6

# Trabalhos Relacionados

Neste capítulo são apresentados alguns trabalhos que de alguma forma possuem relação com o editor gráfico. Boa parte destes trabalhos foram analisados de modo a auxiliar no processo de construção da interface do editor gráfico (ver Seção 3.1).

A escolha destes trabalhos se deve principalmente pela sua disponibilização. Uma parte deles, devido a não ser de domínio público, foi obtida através de programas de demonstração. Os trabalhos escolhidos foram: With Class [40], Rose, BetterState [46, 45], SC e Smart [51].

## 6.1 With Class

With Class é uma ferramenta CASE (*Computer Aided Software Engineering*) desenvolvida pela empresa Microgold Software (Figura 6.1). Seu objetivo é assistir o desenvolvedor de *softwares* na modelagem gráfica de sistemas utilizando metodologias orientada a objetos. As metodologias suportadas pela ferramenta são: OMT [50], Coad-Yourdon, Booch [3] e Shlaer-Mellor.

A ferramenta permite a especificação tanto do modelo estrutural quanto do modelo de controle. Contudo, é possível fazê-lo em uma mesma área de trabalho, misturando conceitos ortogonais. Esse procedimento pode confundir o usuário e complicar a visualização do modelo de objetos gerado.

Com a utilização da ferramenta é possível gerar parte do código que representa o modelo de objetos para as linguagens C++, Eiffel, Ada, Pascal e Smalltalk.

Outras funcionalidades fornecidas pela ferramenta são:

- geração de relatórios através do uso de uma linguagem *script* incorporada à ferramenta;

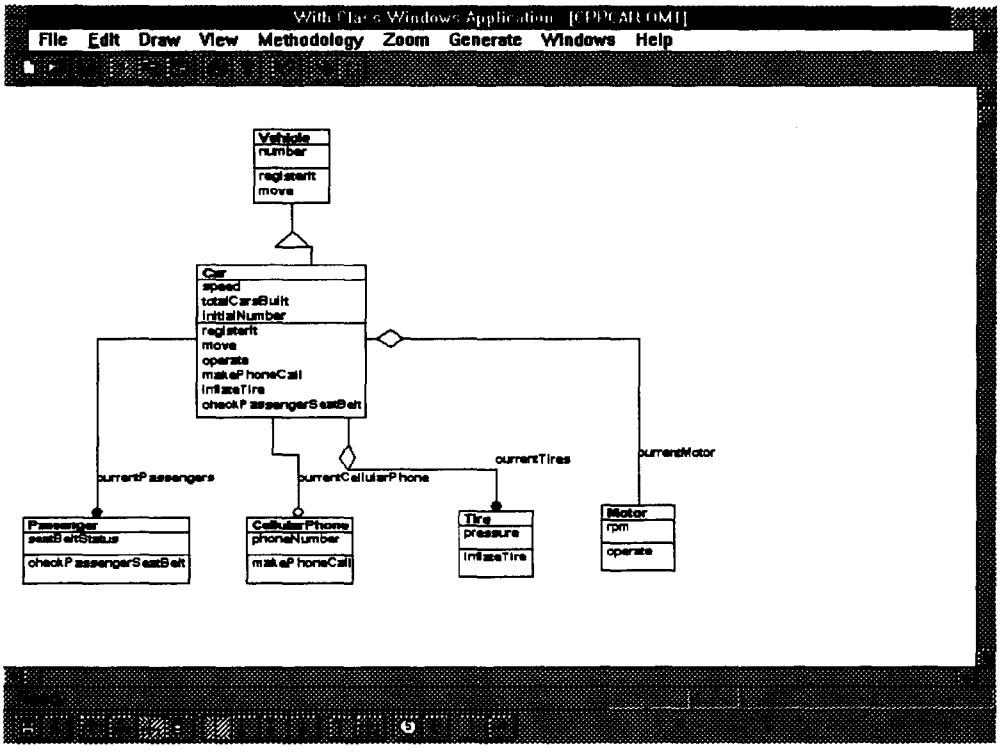



Figura 6.1: Interface da ferramenta With Class

- engenharia reversa, ou seja, permite que se obtenha o modelo de objetos a partir de código fonte;
- geração de dicionários de dados;
- possibilidade de desabilitar a visualização de certas informações do modelo de objetos, tais como: atributos, métodos, relacionamentos, etc.

## 6.2 Rose

Rose é uma ferramenta para construção de programas baseado no paradigma de orientação a objetos desenvolvida pela empresa Rational[1] (Figura 6.2). Rose foi desenvolvida para utilizar a notação UML (*Unified Moddeling Language*) [22], apesar de também suportar as notações OMT [50] e Booch [3].

O editor da ferramenta oferece a possibilidade de se realizar operação de leiaute automático sobre o modelo estrutural gerado, contudo não é permito utilizar essa funcionalidade sobre o modelo de controle. Apesar da disponibilidade desta funcionalidade, os

[1]Maiores informações podem ser encontradas em http://www.rational.com

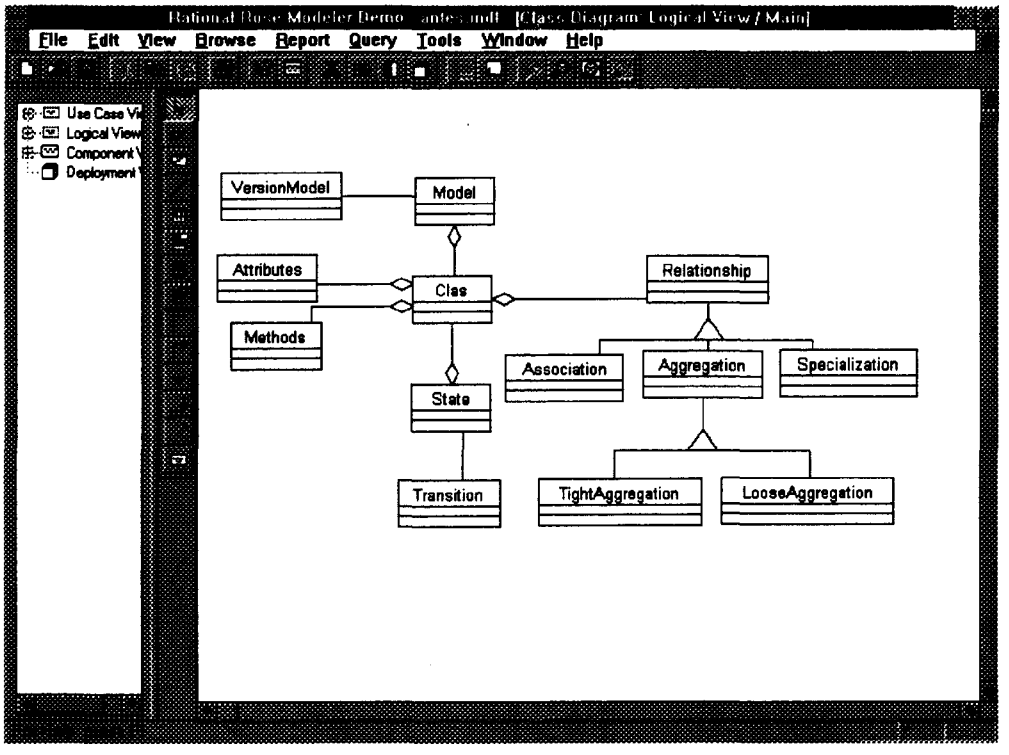



Figura 6.2: Interface da ferramenta Rose

resultados gerados nem sempre são considerados satisfatórios. Na Figura 6.3 é apresentada um exemplo de modelo estrutural construído na ferramenta Rose e utilizando-se o gerador de leiaute automático fornecido pela mesma. O mesmo modelo pode ser visto na Figura 6.4 utilizando, no entanto, o componente Gerador de Leiaute do editor gráfico. Pode-se observar claramente que o leiaute gerado pelo componente Gerador de Leiaute (Figura 6.4) é mais legível do que o gerado pelo editor da ferramenta Rose (Figura 6.3).

A partir da especificação do modelo feito no editor é possível gerar parte do código fonte para as seguintes linguagens de programação: C++, Visual Basic, Java, PowerBuilder, Smalltalk, Forté e Ada.

Rose possui outras funcionalidades além da edição e manipulação de modelos de objetos. São elas:

- operação de *undo* de apenas um nível;
- validação do modelo estrutural criado;
- modelos de controle de versão e facilidades para desenvolvimento em grupo;
- suporte para geração de documentação, inclusive uma descrição textual de todo o modelo;

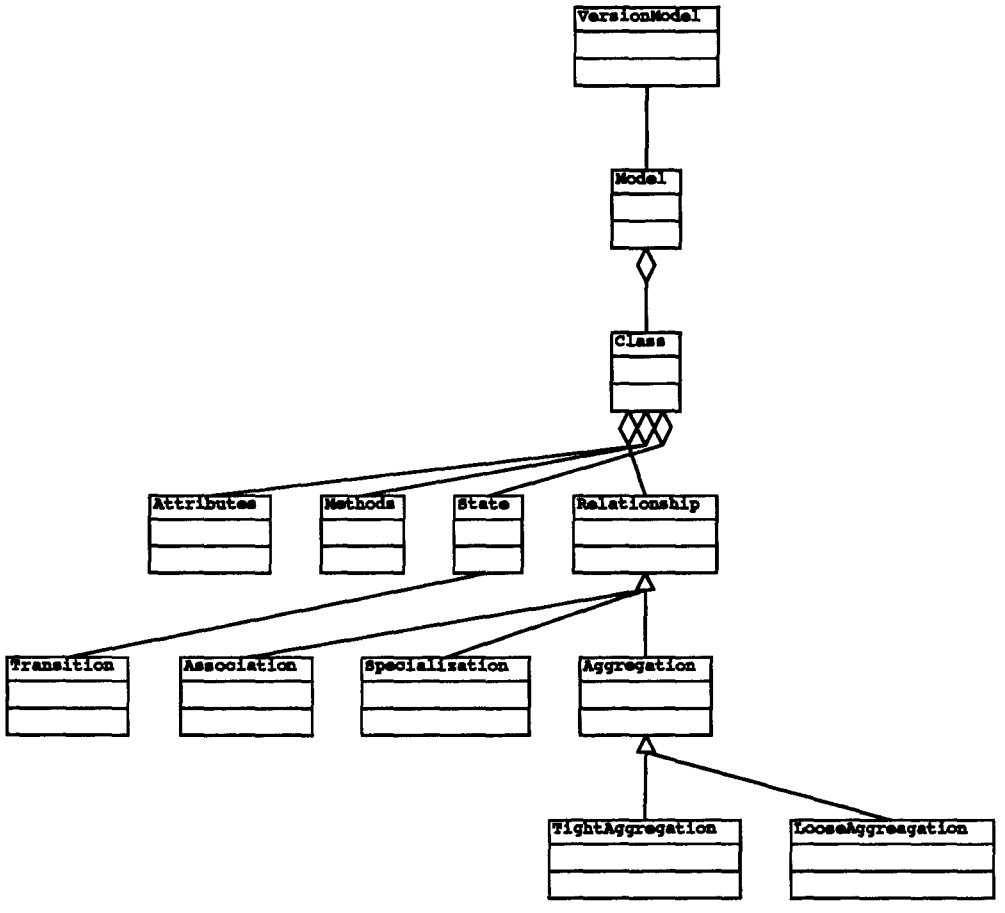



Figura 6.3: Leiaute gerado pelo editor gráfico

- produção de especificações CORBA/IDL para programas distribuídos;
- suporte para ambientes e plataformas heterogêneas;
- capacidade de engenharia reversa.

## 6.3 BetterState

BetterState é uma ferramenta desenvolvida pela empresa R-Active Concepts (Figura 6.5). Seu objetivo é fornecer um ambiente gráfico de modo a facilitar o projeto de software e/ou hardware baseados em máquina de estados. BetterState suporta tanto a linguagem *statecharts* [29] como Redes Petri.

BetterState gera código para C, C++, Java, Delphi, Visual Basic, Perl, VHDL, Verilog HDL, bem como programas cgi's para *web*.

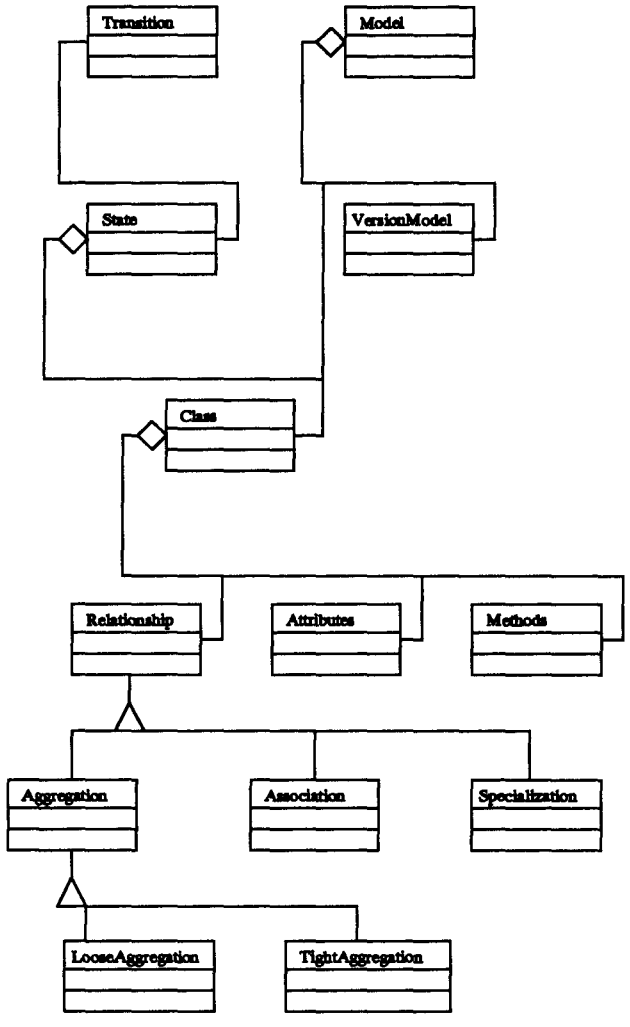



Figura 6.4: Leiaute gerado pela ferramenta Rose

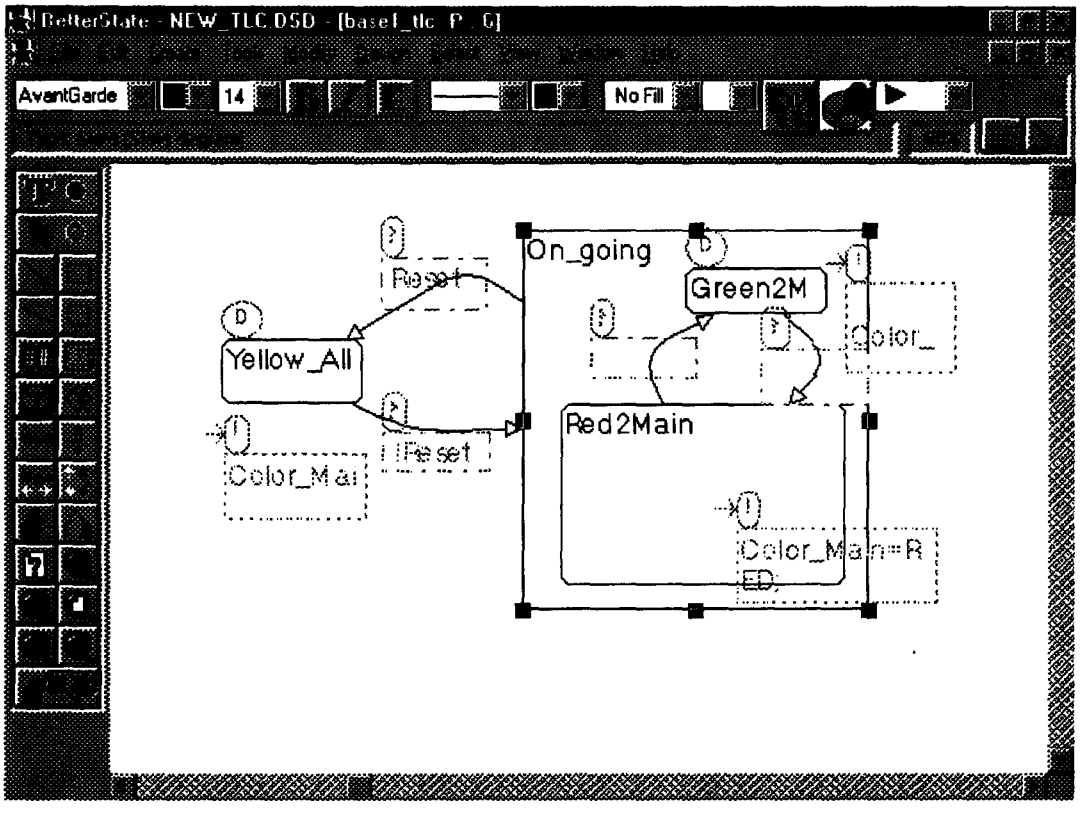



Figura 6.5: Interface da ferramenta BetterState

Algumas das funcionalidades fornecidas pela ferramenta são:

- execução animada da máquina de estados. Com isto é possível verificar, passo a passo, através de uma visualização gráfica, o seu funcionamento;
- operação de *undo* de até dez níveis, além da operação de *redo*;
- quebra da máquina de estados em mais de um diagrama, através de referências entre estados pais e filhos, diminuindo a complexidade do modelo.

## 6.4 SC

SC[2] é um editor gráfico para a linguagem *statecharts* [29] de domínio público desenvolvido por Stephen Edwards na Universidade de Berkeley (Figura 6.6). SC foi baseado em um editor chamado State, desenvolvido para máquina de estados finitos. Contudo,

[2]http://ptolemy.eecs.berkeley.edu/šedwards/statecharts/index.html

State não fornece funcionalidade de hierarquização de estados. O editor SC foi desenvolvido utilizando o *toolkit* Tcl/Tk. Seu projeto e desenvolvimento foi totalmente realizado utilizando-se metodologia orientada a objetos.

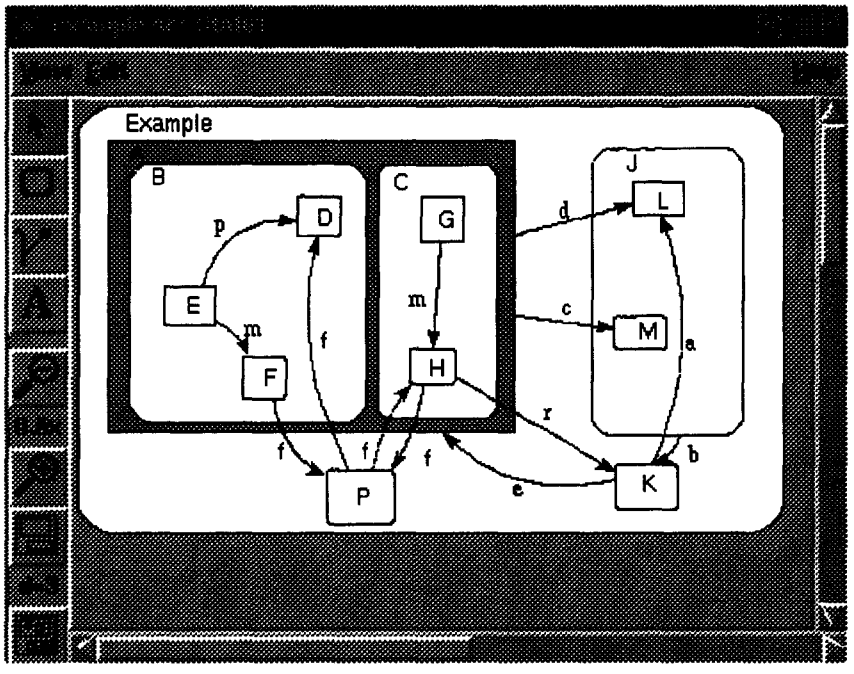



Figura 6.6: Interface do editor gráfico SC

Todas as operações sobre o modelo de *statecharts* são feitas através de manipulação direta. Além das operações de manipulação e edição do diagrama de *statecharts*, o editor fornece as seguintes operações:

- visualizar o modelo de *statecharts* em níveis diferenciados de profundidade de sua hierarquia;
- abrir concorrentemente e de forma sincronizada mais de uma visão do modelo;

Um dos grandes pontos negativos do editor SC é a sua performance, que é considerada muito lenta. Operações simples de edição sobre o modelo são demasiadamente demoradas, o que torna sua utilização bastante desconfortável.

## 6.5 Smart

Smart [51] é um editor gráfico desenvolvido no Instituto de Computação da Universidade Estadual de Campinas como parte integrante do ambiente Xchart [13]. O editor foi criado

com a finalidade de apoiar na especificação de componentes de diálogos de sistemas interativos. Para tanto, baseia-se em uma linguagem de máquina de estados finitos denominada Xchart [12]. A linguagem Xchart permite a hierarquização de estados.

O editor não possibilita a manipulação direta do diagrama de Xcharts. Em contrapartida, fornece duas áreas de trabalho. A primeira, denominada Tree, permite a edição de diagramas de Xcharts através de uma estrutura em árvore. A segunda, denominada XGraph, é utilizada somente para permitir a visualização do diagrama na forma de dígrafo composto.

Smart traduz as representações descritas no diagrama de Xchart para uma linguagem denominada TeXchart (*Textual Xchart*).

O editor também possui a funcionalidades de leiaute automático sobre o diagrama gerado. Para tanto, ele utiliza dois algoritmos de leiaute diferentes para cada uma das áreas de trabalho do editor. Para a área de trabalho *Tree* é utilizado um algoritmo baseado em árvore. E para a área de trabalho Xgraph é utilizado o algoritmo de Sugiyama [53].

## 6.6 Resumo

Neste capítulo, apresentou-se algumas ferramentas para desenvolvimento de aplicações orientadas a objetos, bem como ferramentas para especificação de modelos específicos, tais como *statecharts*. Para cada uma destas ferramentas foram realizados breves comentários de algumas de suas características. Além disso, uma tabela (Tabela 6.1) é apresentada sumarizando algumas destas características.

| Característica | editor gráfico | With Class | Rose | BetterState | SC | Smart |
|---|---|---|---|---|---|---|
| Metodologia | OMT | OMT, Booch, Coad-Yourdon e Shlaer-Mellor | OMT, UML e Booch | Statechart | máquina de estados | Xchart |
| Geração de Código | C++ | C++, Eiffel, Ada, Pascal e Smalltalk | C++, Visual Basic, Power Builder, Smalltalk, Forté, Java e Ada | C, C++, Java, Delphi, Perl, Visual Basic, Verilog HDL, VHDL e cgi | | TeXchart |
| Suporte para Distribuição | √ | | | | | √ |
| Geração de Documentação | | √ | √ | | | |
| Geração de Leiaute | √ | | parcial | | | √ |
| Manipulação Direta | √ | √ | √ | √ | √ | √ |
| Controle de Versão | √ | | √ | | | |
| Engenharia Reversa | | √ | √ | | | |
| Validação de Modelo | | | √ | | | |

Tabela 6.1: Ferramentas de Editoração Gráfica: Uma Comparação

# Capítulo 7

# Conclusões

Como afirmado na Introdução, o trabalho relacionado a esta dissertação envolveu o estudo de diversos tópicos de variadas áreas da Ciência da Computação. Embora, dentre essas áreas, alguns tópicos pudessem ter tido um maior aprofundamento, o desenvolvimento deste trabalho permitiu apontar algumas possíveis contribuições e trabalhos futuro. A seguir, são apresentadas em detalhes cada uma dessas contribuições e trabalhos futuros.

## 7.1 Contribuições

O desenvolvimento de um editor gráfico para um ambiente de programação distribuída orientada a objetos, como descrito neste documento, gerou contribuições principalmente na área de ambientes de programação, mais especificamente na área de programação visual, e na área de algoritmos para traçado e posicionamento de elementos gráfico.

### Ambientes de Programação

Na área de ambientes de programação, podemos citar como contribuição a criação de um ambiente que integra as fases de projeto e implementação de programas distribuídos orientado a objetos. Esta integração traz várias vantagens ao programador, já que minimiza os custos de desenvolvimento, depuração e gerência desses programas. Além disso, diminui a complexidade da programação envolvida, pois estreita a distância entre a abstração utilizada no projeto de programas e a sua implementação. Pesquisas apontam que essa abordagem não tem sido devidamente explorada pela grande parte dos ambientes de programação, o que mostra a viabilidade deste trabalho.

### Algoritmos para Traçado e Posicionamento

Boa parte dos atuais ambientes de programação não fornece nenhum tipo de recurso para auxiliar no traçado e posicionamento automático de elementos gráficos. Os poucos recursos fornecidos por alguns ambiente, na maioria das vezes, produzem resultados que, na maioria das vezes, não correspondem à expectativa. Em conseqüência disto, pode-se citar como uma das contribuições deste trabalho a implementação de um algoritmo para traçado e posicionamento de grafos orientados, voltada especificamente para o traçado e visualização de modelos de objetos.

## 7.2 Trabalhos Futuros

Várias das sugestões para trabalhos futuros apresentadas a seguir são baseadas na experiência acumulada durante o estudo comparativo realizado nos diversos editores gráficos e dos diversos algoritmos para traçado automático de leiautes. Devido ao curto espaço de tempo para desenvolvimento deste trabalho, boa parte destas extensões não foram abordadas.

### Geração de Makefiles

Embora o editor gráfico automatize todo o processo de geração dos metaprogramas, ainda é de responsabilidade do programador criar todo o processo necessário para compilação e ligação dos mesmos. Contudo, essa tarefa também pode ser automatizada, através da geração automática de arquivos *makefiles* (a serem utilizados em conjunto com a ferramenta *make*).

### Validação do Modelo de Objetos

Atualmente muito pouco é feito quanto a verificação da corretude de um modelo de objetos. Assim, seria bastante útil a construção de uma ferramenta capaz de validar modelos de objetos. Ferramentas deste tipo podem auxiliar o programador a identificar construções indevidas à nível de modelo de objetos, o que, conseqüentemente, ajudaria a evitar a geração de programas incorretos.

### Funcionalidades de Edição e Visualização

Além das funcionalidades já existentes, várias outras funcionalidades de edição e visualização podem ser incorporadas ao editor gráfico, dentre as quais pode-se destacar as seguintes:

- **Operação de** *Undo* **e** *Redo*. De modo a possibilitar ao programador desfazer (ou refazer) operações anteriormente realizadas.
- **Operação de** *Zoom*. De modo a permitir o programador controlar o nível de visualização do modelo de objetos.
- **Operação de Cópia, Retirada e Colagem** (*Copy*, *Cut* **e** *Paste*). De modo a permitir a cópia, retirada e colagem de objetos gráficos dos diagramas de modelos de objetos.

Outra funcionalidade de visualização que pode ser adicionada é um mecanismo de *grid*, que pode facilitar bastante nas operações de manipulação do leiaute dos modelos de objetos. Além disto, pode-se integrar esse mecanismo ao componente de Geração de Leiaute, de modo que, caso o programador assim deseje, o componente respeite o valor definido para o *grid* durante o cálculo das posições dos objetos gráficos no leiaute.

## Controle de Profundidade em Hierarquias

À medida que um modelo de objetos cresce, tende a ficar cada vez mais difícil a sua visualização e, conseqüentemente, o seu entendimento. Outros mecanismos, além dos já existentes, podem ser implementados para ajudar ainda mais a redução deste problema. Um deles é a possibilidade do programador controlar o nível de profundidade da hierarquia a ser visualizada no editor em um determinado momento. Esse mecanismo pode ser aplicado ao modelo de controle onde podem existir grande níveis de profundidade na hierarquia de inclusão, o que podem dificultar a visualização do modelo. A utilização deste mecanismo possibilita abstrair os detalhes dos níveis inferiores, focalizando a atenção apenas nas informações dos níveis superiores da hierarquia.

## Fisheye

Além dos mecanismos de visualização referidos acima, uma técnica que também merece ser estudada é o *fisheye* [21]. Esta técnica permite que se visualize detalhes de um determinada figura sem que, no entanto, se perca a visão global da mesma. Para tanto, no caso específico de grafos, são realizadas transformações nas posições dos nós do leiaute do grafo, de modo que uma parte do leiaute seja aumentada, mantendo, entretanto, ainda completamente visível todo o leiaute restante. O uso desta técnica, principalmente no caso de leiaute com grande número de objetos gráficos, permite focalizar detalhes específicos em certos pontos dos modelo de objetos sem que se perda a noção das informações gerais contida no mesmo. Exemplo de um sistema que utiliza esta técnica é o D-ABDUCTOR [55], que é um sistema para visualização e manipulação de dígrafos compostos.

## Algoritmos Geração de Leiautes

Apesar da atual implementação do componente Gerador de Código produzir resultados satisfatórios, ou seja, na maioria dos casos conduzir a leiautes "satisfatórios", diminuindo a complexidade na visualização dos modelos de objetos, há a necessidade de melhorar tais resultados. Uma maneira de obter melhorias neste sentido, seria estudar a criação de novas heurísticas que poderiam ser incorporadas ao algoritmo já implementado. Outra maneira, seria o estudo da viabilidade da implementação de outros algoritmos, além da possibilidade da utilização combinada de mais de um tipo de algoritmo.

Um algoritmo que merece destaque é o algoritmo *Magnetic-Spring* [54]. Este algoritmo pode trazer algumas soluções para problemas encontrados no caso específico do modelo estrutural. Esse algoritmo permite aplicar diferentes campos magnéticos em diferentes conjuntos de arestas. Isto possibilitaria, por exemplo, estabelecer que as arestas representando relacionamentos de agregação e generalização/especialização tendam a ficar na posição vertical, aplicando um campo magnético vertical sobre elas. As arestas que representam relacionamentos de associação, podem ser tratadas com um campo horizontal, ou podem não estar sob influência de um campo. Experimentos devem ser realizados para encontrar a melhor configuração para a aplicação dos campos magnéticos.

Além disso, uma linha de pesquisa dentro da área de Desenho de Grafos, denominada *Dynamic Graph Drawing* [27, 58], merece ser explorada para a representação de modelos de objetos. Essa linha de pesquisa parte do princípio de que é útil ter algoritmos que fazem pequenos ajustes no leiaute, na medida em que o grafo sofre alteração na sua estrutura. O objetivo desta abordagem é tentar convergir o leiaute para um estado considerado "bom", fazendo pequenas melhorias de "embelezamento", sem que, no entanto, isto provoque uma drástica modificação no leiaute original, o que poderia causar a perda da localização das informações no leiaute por parte do observador. Este problema é conhecido originalmente por *mental map*.

## Mecanismo de Controle de Versões

Outro ponto que ainda pode ser bastante melhorado é o mecanismo de controle de versões. Durante o início dos trabalhos, não se tinha como objetivo a implementação de nenhum tipo de controle de versões. Contudo, durante o desenvolvimento do mesmo, observou-se que a implementação de um mecanismo simples, como o que foi implementado, poderia trazer benefícios satisfatórios, sem no entanto resultar em um aumento considerável de trabalho, o que provocaria desvios nos objetivos principais do trabalho. Contudo, para um melhor aproveitamento desta funcionalidade há a necessidade da implementação de um mecanismo mais eficiente e de melhor qualidade.

# Referências Bibliográficas


[1] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data-flow diagrams. *IEEE Trans. Softw. Eng.*, SE-12(4):538–546, 1986.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Elsevier North Holland, Inc., New York, 1979.

[3] Grady Booch. *Object Oriented Design With Applications.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, 1991.

[4] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* PhD thesis, Dept. of Computing Science, University of Newcastle upon Tyne, october 1994.

[5] K. M. Chandy and J. Misra. *Parallel Program Design*, pages 289–312. Addison-Wesley Publishing Company, 1988.

[6] N. Chiba, K. Onoguchi, and T. Nishizeki. Drawing planar graphs nicely. *Acta Inform.*, 22:187–201, 1985.

[7] S. Chiba. A Metaobject Protocol for C++. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95)*, pages 285–299, 1995.

[8] H. Thomas Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 477–485,488–494. MIT Press and McGraw-Hill, Cambridge, Massachusetts, first edition, 1993. ISBN=0-262-03141-8 (MIT Press), ISBN 0-07-013143-0 (McGraw-Hill).

[9] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, october 1996.

[10] F. N. de Lucena and H. K. E. Liesenberg. Interfaces Homem-Computador: uma primeira introdução. Technical Report 94-07, Instituto de Computação, Universidade Estadual de Campinas, 1994.

[11] F. N. de Lucena and H. K. E. Liesenberg. Fundamentos de Interfaces Homem-Computador, 1996. http://www.dcc.unicamp.br/projects/Xchart/preparacao.html.

[12] F. N. de Lucena, H. K. E. Liesenberg, and L. E. Buzato. Especificação da Linguagem Xchart. Technical report, Instituto de Computação, Universidade Estadual de Campinas. http://www.dcc.unicamp.br/proj-xchart/preparation.html.

[13] F. N. de Lucena, H. K. E. Liesenberg, and L. E. Buzato. Xchart-Based Complex Dialogue Development. In *Simpósio Nipo-Brasileiro de Ciência e Tecnologia*, pages 387–396, august 1995. http://www.dcc.unicamp.br/proj-xchart/publications.html.

[14] Departament of Computing Science, University of Newcastle upon Tyne. *The Arjuna System Programmer's Guide*, public release 3.0 edition.

[15] G. Di Batista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing Directed Acyclic Graphs: An Experimental Study. Technical Report CS-96-24, Department of Computer Science, Brown University, october 1996.

[16] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. Preprint, Dept. Comput. Sci., Brown Univ., Providence, RI, november 1993. To appear in *Comput. Geom. Theory Appl.* Preliminary version available via anonymous `ftp` from `wilma.cs.brown.edu`, `gdbiblio.tex.Z` and `gdbiblio.ps.Z` in `/pub/papers/compgeo`.

[17] E. W. Dikjstra. Two Starvation Free Solutions to a General Exclusion Problem. Technical Report EWD 625, 1978.

[18] H. A. D. do Nascimento. Uma Abordagem para o Desenho de Grafos Baseada na Utiliza,c ao de Times Ass'incronos. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, may 1997.

[19] P. Eades and T. Lin. Algorithmic and declarative approaches to aesthetic layout. In *Graph Drawing '93 (Proc. ALCOM Workshop on Graph Drawing)*, Paris, France, september 1993.

[20] M. A. Ellis and B. Stroustrup. *C++: manual de referência comentado.* Ed. Campus, Rio de Janeiro, 1993.

[21] A. Formella and J. Keller. Generalized fisheye views of graphs. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 242–253. Springer-Verlag, 1996.

[22] M. Fowler. *UML Distilled: applying the standard object modeling language.* Addison Wesley Longman, Inc., 1997.

[23] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system *davinci.* In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 266–269. Springer-Verlag, 1995.

[24] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. - Pract. Exp.*, 21(11):1129–1164, 1991.

[25] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

[26] M. R. Garey and D. S. Johnson. *Computer and Intractability: a guide to the theory of NP-Completeness*, page 192. W. H. Freeman and Company, 1979.

[27] Ashim Garg and Roberto Tamassia. Advances in graph drawing. In *Algorithms and Complexity (Proc. CIAC'94)*, volume 778 of *Lecture Notes in Computer Science*, pages 12–21. Springer-Verlag, 1994. ftp://wilma.cs.brown.edu/pub/papers/compgeo/gdadv-ciac94.ps.Z.

[28] D. J. Gschwind and T. P. Murtagh. A recursive algorithm for drawing hierarchical directed graphs. Technical Report CS-89-02, Department of Computer Science, Williams College, 1989.

[29] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, june 1987.

[30] D. Harel and M. Sardas. Randomized graph drawing with heavy-duty preprocessing. *J. Visual Lang. Comput.*, 6(3), 1995. (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).

[31] H. R. Hartson and D. Hix. Human-Computer Interface Development: concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, march 1989.

[32] R. Hartson. User-Interface Management Control and Communication. *IEEE Software*, january 1989.

[33] M. Himsolt. GraphEd: a graphical platform for the implementation of graph algorithms. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 182–193. Springer-Verlag, 1995.

[34] T. Kamada and S. Kawai. Automatic display of network structures for human understanding. Technical Report 88-007, Department of Information Science, University of Tokyo, 1988.

[35] R. N. Lacerda. Uma Breve Análise Sobre Alguns Toolkits, december 1995. Monografia apresentada como parte dos trabalhos relacionados com a disciplina MO622 (Fatores Humanos em Sistemas de Computação).

[36] I. K. Lam. *Tix Programming Guide.* Expert Interface Technologies, 1996. ftp://ftp.xpi.com/pub/tix-4.0.ps.gz.

[37] C. Lewis and J. Rieman. *Task-Oriented User Interface Design: a pratical introducion.* Shareware, 1994. FTP anônimo: ftp.cs.colorado.edu, /pub/CS/distribs/clewis/HCI-Design-book.

[38] P. Maes. Concepts and Experiments in Computational Refletion. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 147–155, 1987.

[39] E. B. Messinger, L. A. Rowe, and R. H. Henry. A divide-and-conquer algorithm for the automatic layout of large directed graphs. *IEEE Trans. Syst. Man Cybern.*, SMC-21(1):1–12, 1991.

[40] MicroGold Software, Inc. *With Class 2.5 - User Manual*, 1995.

[41] J. K. Ousterhout. *Tcl and Tk Toolkit.* Addison-Wesley Publishing Company, Inc., april 1994. ISBN=0-261-63337-X.

[42] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna.

[43] L. B. Protsko, P. G. Sorenson, J. P. Tremblay, and D. A. Schaefer. Towards the automatic generation of software diagrams. *IEEE Trans. Softw. Eng.*, SE-17(1):10–21, 1991.

[44] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 435–446. Springer-Verlag, 1996.

[45] R-Active Concepts, Inc. *BetterState Pro - An Introdution to Design with Statecharts*, 1996.

[46] R-Active Concepts, Inc. *BetterState Pro - Getting Started*, 1996.

[47] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Softw. - Pract. Exp.*, 17(1):61–76, 1987.

[48] J. Rumbaugh. OMT: the dynamic model. *JOOP*, 7(9), february 1995.

[49] J. Rumbaugh. OMT: the object model. *JOOP*, 7(8), january 1995.

[50] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *The Oriented Object Modeling and Design.* Pretince-Hall International, Inc., Englewood Cliffs, NJ, USA, 1992.

[51] O. Severino Junior. Smart: Um Editor Gráfico de Diagramas em Xchart. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, december 1996.

[52] Sidney L. Smith and Jane N. Mosier. Guidelines for Designig User Interface Software. Technical Report ESD-TR-86-278, US Air Force, 1986.

[53] K. Sugiyama and K. Misue. Visualizing Structural Information: automatic drawing of a compound digraph. *IEEE Transaction on Systems, Man & Cybernetics*, 21(4):876–892, july/august 1991.

[54] K. Sugiyama and K. Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375. Springer-Verlag, 1995.

[55] K. Sugiyama and K. Misue. A generic compound graph visualizer/manipulator: D-abductor. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 500–503. Springer-Verlag, 1996.

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

[57] J. L. Szwarcfiter. *Grafos e Algoritmos Computacionais.* Editora Campus, Rio de Janeiro, 1986.

[58] R. Tamassia. Graph drawing. In Eli Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry.* CRC Press. to appear.

[59] R. Tamassia, G. Di Batista, and C. Batini. Automatic Graph Drawing and Readability of Diagrams. *IEEE Transactions on Systems, Man & Cybernetics*, 18(1):61–79, february 1988.

[60] A. S. Tanembaum. *Modern Operating Systems*, pages 41–43,485–498. Pretince-Hall International, Inc., Englewood Cliffs, NJ, 1992.

[61] D. Tunkelang. A pratical approach to drawing undirected graphs. Technical Report CMU-CS-94-161, School of Computer Science, Carnegie Mellon University, 1994.

[62] B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall, 1995.