UMA MODELAGEM E IMPLEMENTAÇÃO DO PRINCÍPIO DE MÚLTIPLAS VISÕES COM REATIVIDADE EM C++

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Daniel Tavares Correia Xavier e aprovada pela Comissão Julgadora.

Campinas, 4 de fevereiro de 1992

Prof. Dr. Hans Kurt Edmund Liesenberg
Orientador

Dissertação apresentada no Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de Mestre em Ciência da Computação.

Tem alguma idéia de por quantas vidas tivemos de passar até chegarmos a ter a primeira intuição de que há na vida algo mais do que comer, ou lutar, ou ter uma posição importante dentro do bando? (...) escolheremos o nosso próximo mundo através daquilo que aprendemos neste. Não aprender nada significaria que o nosso mundo será igual a este, com as mesmas limitações e pesos de chumbo a vencer.

Richard Bach em
A História de Fernão Capelo Gaivota

Agradecimentos

O esforço para elaboração deste trabalho contou com o apoio direto e indireto de diversas pessoas sem as quais a dissertação não teria sido concluída.

Gostaria de agradecer, em primeiro lugar, ao meu orientador, Prof. Hans Kurt Edmund Liesenberg pela confiança e apoio técnico dispendido durante toda a fase de pós-graduação; a Frederico Sidney Cox (DCC-UNICAMP) por estar sempre disposto a discutir sobre o tema da tese e a Welson Régis Jacometti (DCC-UNICAMP) por sempre socorrer nos *bugs* do XView.

Ao grupo criador do ET++ da Universidade de Zürich, em particular à Erich Gamma, por remeter pessoalmente as últimas versões da documentação do ET++.

Ao pessoal da Sun Microsystems no Brasil, por fornecer o compilador C++ da AT&T release 2.0.

Gostaria de agradecer também ao pessoal da secretaria do DCC-IMECC, em particular a Ademilsom Camargo, Solange Mendes e Luíz Rosa, pela cooperação na retaguarda administrativa.

Agradecimentos pessoais são dirigidos ao meu companheiro e amigo Dr. Giovani Machado por me ter "aturado" durante todo este desfecho final da tese; aos amigos e "irmãos" Aluízio Araújo e Ana Cristina Fernandes, que mesmo a 10.000 km de distância, estiveram sempre tão presentes, e aos meus pais Lívio Xavier e Lúcia Tavares, pela amizade, confiança e apoio incondicional dado nestes últimos anos.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), ao Conselho Nacional de Pesquisa e Desenvolvimento Tecnológico (CNPq) e à Universidade Estadual de Campinas (UNICAMP) pelo apoio financeiro dado durante o período de pós-graduação.

Finalmente, gostaria de agradecer às pescarias e banhos de mar em Tamandaré (litoral sul de Pernambuco), que não pude realizar e assim, graças a essa impossibilidade, o presente trabalho tornou-se uma realidade.

Considerações Iniciais

A linguagem utilizada na redação deste texto é bastante peculiar. A tradução de alguns termos para o português "tentou" manter uma certa fidelidade ao original. Por exemplo, *user interface* foi traduzida para interface-usuário, e em alguns casos, para interface homem-máquina; *event handler* para gerenciador de eventos, etc. Entretanto, para algumas palavras, decidimos optar pelos próprios termos originais, tais como: *mouse*, *toolkit*, *menus*, *drag*, *click*, etc., por não existir, na língua portuguesa, uma tradução equivalente dos mesmos (ou pelo menos aceita universalmente).

Por outro lado, alguns termos **inexistentes** na língua portuguesa foram amplamente e **propositadamente utilizados** (outros não) no texto (é o caso da palavras **instanciação** e **inicialização**), por terem um significado "conhecido" para o pessoal ligado ao estilo de programação orientado a objetos. Em todo o texto, os termos que não sofreram traduções foram rotulados como tal através do fonte em *itálico.*

Algumas convenções tipográficas foram utilizadas, a saber:

- *itálico* - termos em língua estrangeira
  ex. *toolkit*, *mouse*, *menu*, etc.
- **negrito** - nome de capítulos, seções, subseções e expressões ou termos cujo conteúdo tem importância fundamental para a compreensão do contexto
  ex. **capítulo 3, polimorfismo**
- sans serif - nomes de produtos ou softwares, nomes de empresas e termos patenteados
  ex: XView Toolkit, ET++, C++, Sun Microsystems, etc.
- `typewriter` - exemplos de programas e comandos de linguagens.
  ex: `MyView->Draw();`

Conteúdo

Prefácio

Esta dissertação discute aspectos relacionados com o uso e a implementação de interfaces homem-máquina sob o ponto de vista do projetista de software. É feito um breve relato sobre a evolução do processo de interação entre o homem e o computador desde o surgimento das interfaces seqüênciais até as interfaces assícronas com interação através de manipulação direta. É implementada uma proposta de um modelo de comunicação entre aplicações e usuário final chamado de **princípio de múltiplas visões com reatividade** ou simplesmente **princípio reativo** que pode ser definido como sendo as diferentes formas pelas quais os componentes ou variáveis de um sistema computacional podem ser **vistos** e **estimulados**. Para implementação do princípio reativo utilizou-se como base de desenvolvimento o **paradigma de programação orientado a objetos** dada as facilidades providas para modelar interfaces homem-máquina. Diversas entidades desde uma janela representando a aplicação, até os *menus* e botões representando as opções disponíveis em um sistema podem ser facilmente "abstraídos" como objetos. Em particular, no princípio reativo, as variáveis e os componentes do sistema sendo desenvolvido são ditos **objetos interativos** do sistema. E, devido a este fato, eles possuem **estado** e **comportamento** próprios cujos valores internos devem ser visíveis e estimuláveis. A linguagem base adotada na implementação do princípio reativo foi a C++ e o ambiente sobre o qual ele foi desenvolvido foi o XView, um *toolkit* implementado sobre o sistema XWindow.

A dissertação foi dividida em cinco capítulos e dois apêndices:

- o capítulo 1 relata a utilização de interfaces homem-máquina em sistemas computacionais. Faz um breve histórico da evolução do processo de interação culminando com a utilização dos sistemas de gerência de interface com o usuário;
- o capítulo 2 diz respeito à linha de implementação adotada para construção do princípio reativo: o paradigma de programação orientado a objetos. Para introdução dos principais conceitos do paradigma, são mostrados exemplos em C++: a linguagem de implementação adotada;
- o capítulo 3 descreve, superficialmente, o *toolkit* orientado a objetos utilizado como base para a modelagem e a implementação do problema proposto e também a maneira encontrada para viabilização da interligação do código do C++ com funções

do XView;

- o capítulo 4 relata a especificação do problema implementado bem como a descrição detalhada do seu propósito, evolução e simplificação de utilização por parte dos usuários finais. É descrita, também, a implementação, aspectos externos e a hierarquia de abstrações utilizada para modelar o princípio reativo.

- O capítulo 5 enumera algumas conclusões tiradas da fase de especificação e implementação do problema proposto. Ele enumera ainda experiências negativas obtidas na tentativa de adoção de outros sistemas de geração de interfaces (o ET++) e experiências com o próprio sistema *toolkit* utilizado;

- o apêndice A apresenta a especificação de todos os componentes interativos implementados na construção do princípio reativo. É mostrado ainda a especificação das classes abstratas e de controle definidas na hierarquia de abstração do problema implementado;

- o apêndice B descreve o `DrawCircuitTool`, um simulador de circuitos lógicos implementado para servir de exemplo prático através do qual o princípio reativo poderia ser demonstrado. É descrita também a maneira como é feita a incorporação do princípio de múltiplas visões com reatividade em aplicações com naturezas diversas.

Capítulo 1

Introdução

1.1 Motivação

"O grande passo para uma utilização mais abrangente dos computadores seria o manuseio destes por profissionais outros, não relacionados diretamente à área de Ciência da Computação!" Esta afirmativa nos leva a crer que a "popularidade" e a desmitificação dos computadores poderia ser uma realidade se o uso destas máquinas perdesse a característica de atividade inacessível, com a ajuda não só dos ditos usuários finais, como também pelos programadores das aplicações. Este processo de "aproximação" entre o homem comum e o computador está, inevitavelmente, relacionado à elaboração de programas computacionais mais amigáveis e representa uma vertente importante de pesquisa denominada **comunicação entre o homem e o computador.**

Pode-se assegurar, hoje em dia, que a limitação de uso de um computador ou de um sistema computacional não reside apenas no hardware que o mesmo possua, mas principalmente na "força" da sua comunicação com os usuários. Esta afirmativa poderia ser completamente infundada se houvesse sido formulada há alguns anos atrás, quando a grande preocupação era apenas a de saber se o tamanho da memória era de 1 ou 2 Megabytes ou se o disco rígido da máquina suportaria uma determinada aplicação. Desta forma, questionamentos do tipo: "Este sistema pode ser executado em uma Sun Sparc Station[1] com processador RISC (Reduced Instruction Set Computer) ou em um AT-386 com *clock* de 33 Megahertz", deveriam ser reformulados de molde a incluir a filosofia da construção da interface homem-máquina adotada tal como será visto.

Apesar da comunicação entre homem e computador ter uma importância fundamental no desenvolvimento de sistemas, apenas no final da década de 1980 é que o **desenvolvimento de interfaces** começou a tornar-se objeto de pesquisas. Um fato que se mostra essencial na construção de software parece ser cada vez mais o desenvolvimento da interface **isolada** do desenvolvimento do sistema propriamente dito. Trabalhos recentes nestas

[1]Sun Sparc Station é marca registrada da Sun Microsystems Inc.

áreas [Fei87, HH89] são unânimes em afirmar que: "Interfaces homem-máquina são uma área de pesquisa de grande impacto e com taxas de crescimentos crescentes".

Não somente aspectos técnicos são utilizados para se chegar a boas interfaces, como por exemplo o formalismo a ser adotado para a efetivação do diálogo ou o tipo de *menu* a ser adotado para o desenvolvimento do sistema, mas também aspectos relacionados com o usuário final ou os aspectos que dizem respeito aos **fatores humanos** das interfaces: a **ergonomia**, a **psicologia cognitiva**, as **artes visuais** e a **psicologia comportamental**; áreas que relacionam os aspectos mais "humanos" da construção do software, como por exemplo: a cor a ser adotada como *background* de *menus* ou aspectos relacionados à capacidade do usuário responder mais rapidamente aos questionamentos do sistema[2] ou ainda as relações entre o homem e o seu ambiente de trabalho, dentre outros, como será sucintamente citado a seguir, já que este tema não faz parte do objeto principal do trabalho.

1.2 Terminologia básica

Com o objetivo de tornar clara a discussão sobre interfaces, alguns termos e definições devem ser elucidados para evitar ambigüidades nas interpretações.

Os termos **diálogo homem-máquina**[3], e **interface homem-máquina**[4] representam conceitos diferentes, a saber:

- diálogo homem-máquina denota a comunicação entre o ser humano homem e um sistema computacional, ou ainda: "a troca bi-direcional de símbolos e mensagens entre o homem e o computador";
- interface homem-máquina denota o agente intermediário utilizado para efetivação desta comunicação, ou seja, é o software e o hardware utilizado para efetivar a troca de comunicação entre o homem e a máquina.

Gerenciamento de diálogos diz respeito a aspectos de desenvolvimento de interfaces, e abrangendo: representação, análise, projeto, implementação, execução, evolução e manutenção.

Sistemas de Gerência de Interface com Usuário (SGIU)[5] podem ser resumidos como sistemas ou ferramentas que suportam o desenvolvimento de interfaces, utilizando as atividades citadas para o gerenciamento de diálogos. Nestas ferramentas, como será descrito, detalhes de implementação estão isolados dos aspectos de construção de software. O SGIU gera aplicações interativas de acordo com a especificação fornecida pelo projetista.

[2] aprimoramento do impulso do reflexo.
[3] do inglês *human-computer dialogue*.
[4] do inglês *human-computer interface* ou *user interface*.
[5] do inglês User Interface Manegement Systems - UIMS.

Esta especificação pode ser uma descrição textual da interface; para isto, podem ser usados formalismos já existentes como, por exemplo, a BNF (Backus Normal Form)[6], Autômatos Finitos, Diagramas de Transição de Estados, dentre outros.

1.3 Interfaces homem-máquina e seu gerenciamento

Interfaces homem-máquina são a parte de um sistema computacional que permite que o usuário possa interagir com o computador. Esta interação pode se dar, por exemplo, em um determinado ambiente, através da entrada, armazenamento, manipulação e da busca de dados. Uma interface bem definida faz com que o usuário possa ter um acesso rápido e seguro ao núcleo[7] de um sistema. Além de uma manipulação geral de dados, uma interface pode oferecer ainda opções para disparar processos, modificar aspectos visuais da apresentação, selecionar opções e, principalmente, permitir que o usuário possa tomar decisões sobre sistemas cujo desenvolvimento é apoiado por computador. Analogamente, uma interface mal definida faz com que a tomada de ações ou a seleção de opções seja feita de maneira mais lenta e conseqüentemente não confiável em relação a um sistema computacional.

As interfaces homem-máquina podem ser divididas em dois grupos: as não-interativas e as interativas. Nas interfaces não-interativas uma eventual entrada de dados precisa ser preparada, de uma forma textual, e submetida ao sistema, que por sua vez irá validar, processar e depois mostrar os resultados, caso existam. Todo este processo leva a crer que a sua utilização freqüente é bem mais lenta e bem mais difícil se relacionada às interfaces interativas.

As interfaces interativas por sua vez caracterizam-se por permitir aos usuários um retorno[8] mais rápido e eficiente as suas consultas. As interfaces interativas são consideradas mais "populares" e conseqüentemente são alvos de maiores pesquisas se comparadas às interfaces não-interativas. A razão disto é o fato da tecnologia atual de componentes possuir dispositivos de entrada/saída, *i.e.*, *mouse*, monitores gráficos, dentre outros, que provêem facilidades para suportar interações rápidas e eficientes com a aplicação.

Segundo Hartson [HH89], em sua revisão sobre o desenvolvimento de interfaces homem-máquina, "a discussão sobre interfaces interativas e o seu gerenciamento está implicitamente relacionada com a descrição dos tipos de diálogos que ela pode oferecer". Segundo ele, existem dois tipos de metáforas que descrevem como o homem e o computador trocam informações: a metáfora do **mundo conversacional** (*conversational world*), que está associada ao diálogo seqüencial, e a metáfora do **mundo modelo** (*model world*), relacionada esta com diálogos assíncronos. A figura 1.1 ilustra as duas metáforas associadas com os

[6] será visto que, na realidade, são usadas extensões ou adaptações da forma original de BNF com variações que possam refletir aspectos mais semânticos de comunicações entre o homem e a máquina.

[7] por núcleo entenda-se principais recursos que o sistema possa oferecer.

[8] no sentido da palavra inglesa *feedback*.

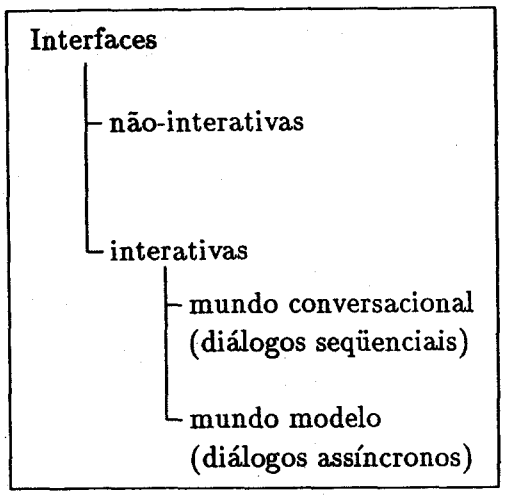

Figura 1.1: Interfaces e metáforas segundo Hartson [HH89]

tipos de interface.

No mundo conversacional, o usuário descreve o que fazer, usando geralmente comandos. O diálogo para este tipo de metáfora é o **diálogo seqüencial**. Aqui, o usuário movimenta-se de uma parte do diálogo para a seguinte, dando uma idéia de seqüencialidade de comandos. O **diálogo seqüencial** pode incluir movimentação através de uma rede de *menus*, entrada de dados e interações do tipo pergunta/resposta. Uma vantagem deste tipo de diálogo é a possibilidade de o usuário poder visualizar a seqüência lógica de comandos fornecida e o comportamento do sistema em relação a eles. Um exemplo de um diálogo seqüencial é mostrado na figura 1.2 onde podemos ver a tela de um microcomputador executando um compilador Turbo C Versão 2.0[9]. Neste exemplo, é possível visualizar a seqüência de passos do diálogo através do percurso que foi realizado sobre a rede de *menus*. O usuário selecionou primeiro a opção `Options` do *menu* principal, que produziu um *menu pulldown* com as opções: `Compiler`, `Linker`, `Environment`, `Directories`, `Arguments`, `Save Options` e `Retrieve Options`. Em seguida a opção `Compiler` foi selecionada e fez com que um novo *menu* fosse exibido com as opções: `Model`, `Defines`, `Code Generation`, `Optimization`, `Source`, `Errors` e `Names`. Neste ponto, o usuário selecionou a opção `Code Generation` que fez com que um terceiro *menu* fosse apresentado. Por fim, o usuário modificou a opção `Test Stack Overflow` de `On` para `Off`.

Toda esta seqüência de ações pode ter um efeito bastante exaustivo, do ponto de vista do usuário, quando da utilização sistemática deste tipo de diálogo. Desta forma, algumas variações foram sendo realizadas sobre este tipo de interação, como por exemplo a

[9]Turbo C é marca registrada da Borland International.

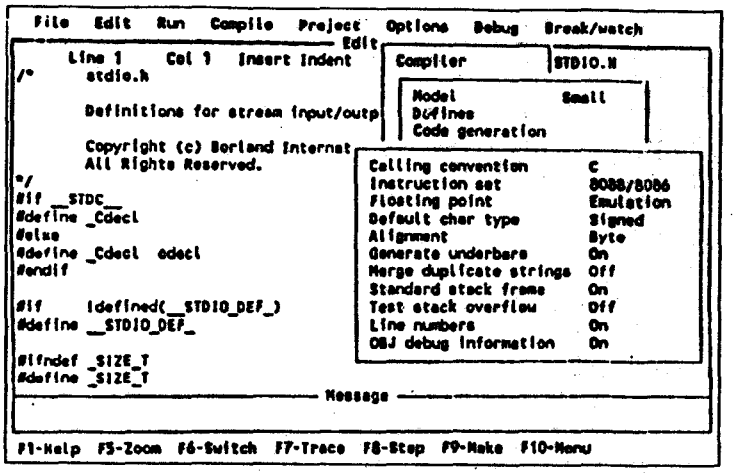

Figura 1.2: Exemplo de um diálogo seqüencial

introdução das chamadas *hot-keys*, que possibilitam a ruptura da seqüência de uma determinada ação e a ativação desta através de um único comando. Na figura 1.3 temos, por exemplo, a efetivação do comando de finalização do programa tanto a partir da seqüência de *menus*: `File - Quit`, ou a partir do simples acionamento das teclas **`Alt-X`**.

No **mundo modelo**, por sua vez, o usuário descreve as suas ações geralmente movimentando e selecionando estruturas através do *mouse* ou de um outro dispositivo de apontar para as representações visuais de objetos. Este tipo de interação também pode ser chamado de **manipulação direta**; o diálogo a ele associado é denominado **diálogo assíncrono**, já que não há sincronismo e a seqüência contínua de comandos presente, como no modelo anterior, tal como será melhor detalhado. A figura 1.4 ilustra a ativação (ou seleção) de uma figura num sistema de elaboração de desenhos do tipo PC-Paintbrush[10] usando a manipulação direta. Na parte (a) é mostrada a posição e o tamanho original da referida janela e, na parte (b), o usuário apontou e selecionou um dos "handles"[11] da janela e movimentou o *mouse* para a nova posição desejada, redimensionando o tamanho original.

O diálogo associado ao mundo modelo é o diálogo assíncrono ou não seqüencial, como já dito. No diálogo seqüencial do mundo conversacional, o usuário interage com o sistema "respondendo" a uma pergunta de cada vez, de forma seqüencial. Já no **diálogo assíncrono**, diversos questionamentos podem estar disponíveis em um determinado ins-

[10] PC-Paintbrush é marca registrada da ZSoft Corporation.

[11] no sentido literal da palavra: ou seja, os manipuladores da janela funcionam como uma espécie de "alça" que permitem a movimentação e o redimensionamento de objetos.

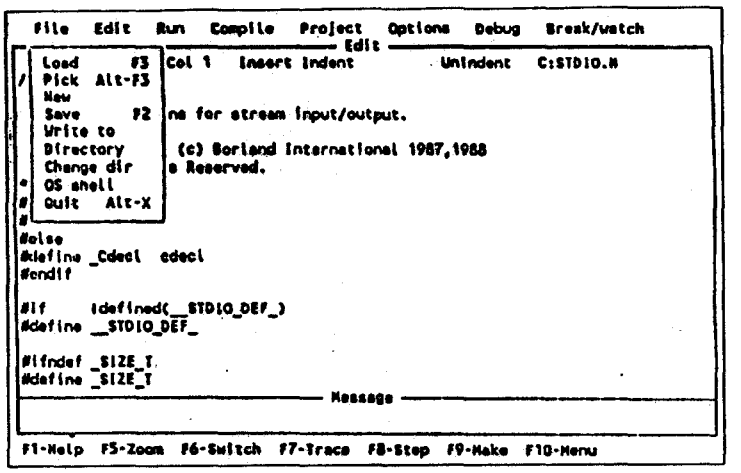

Figura 1.3: Diálogo seqüencial com *hot-keys*

tante. O usuário, por sua vez, pode interagir com qualquer uma das tarefas[12] disponíveis de uma forma não seqüencial. Estas interações **devem** ser independentes umas das outras. Na figura 1.5 é mostrada uma tela de um editor de textos do tipo **WYSIWYG** (What You See Is What You Get) representando um diálogo assíncrono. O diálogo não seqüencial ou assíncrono é também conhecido como **diálogo dependente de eventos**[13], pois o simples *"click"* do *mouse* em um ícone faz com que uma ação seja interpretada como um evento de entrada. Na figura 1.5, por exemplo, temos seis tarefas assíncronas representando a ação `Break` do editor: `Page-Break`, `Column-Break`, `Line-Break`, `Next-Y-Position`, `Allow Within` e `Keep With Next`. O simples *"click"* do *mouse* em uma dessas tarefas faz com que a ação seja interpretada como um evento de entrada e os processos associados àquela opção sejam, então, executados.

No Capítulo 4, que trata da descrição do **processo de múltiplas visões com reatividade**, veremos que ao diálogo assíncrono estão associados outros tipos de diálogos, como por exemplo o **diálogo concorrente**. Neste caso, poderemos ter na tela, para efeito de ilustração deste diálogo, um relógio sendo representado por duas visões: a analógica e a digital, conforme a figura 1.6. A atualização dos valores em uma destas visões deve refletir na outra e vice-versa, dando uma "idéia" de concorrência de processos na atualização de valores.

Uma correspondência grosseira entre o mundo conversacional e o mundo modelo poderia ser exemplificada assim: Um usuário, no **mundo conversacional**, para modificar ou selecionar uma opção a ser executada, interage com a máquina, geralmente via teclado,

[12] do inglês *task*.

[13] do inglês *event-based dialog*.

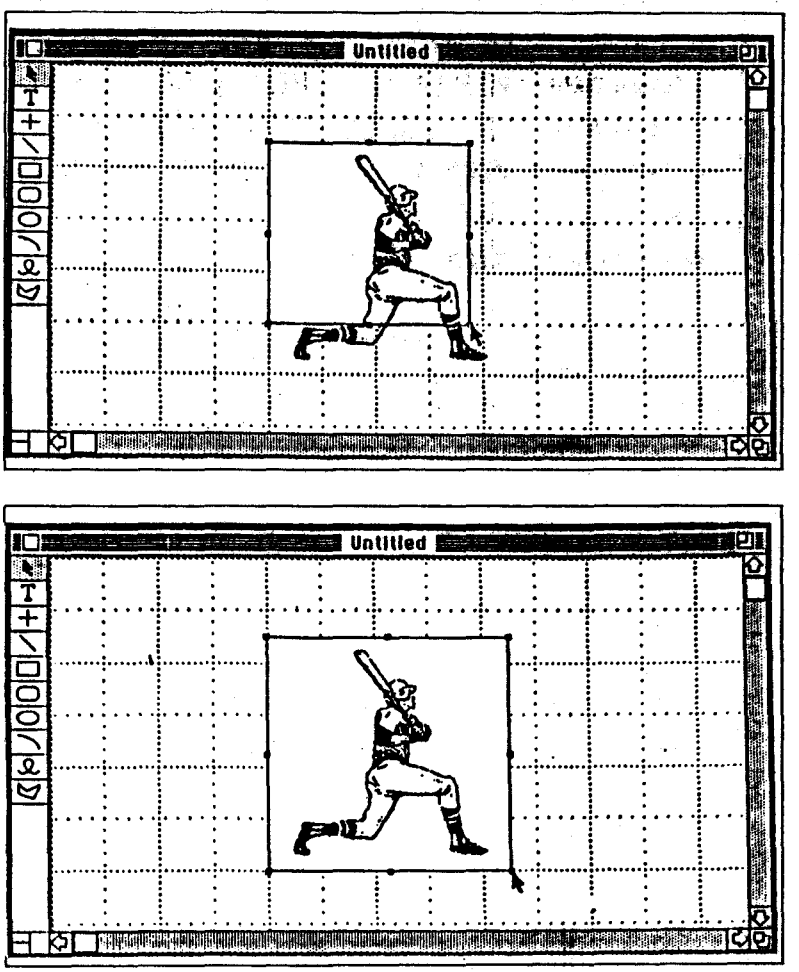

Figura 1.4: Exemplo de um redimensionamento de janelas utilizando o diálogo com manipulação direta: (a) posição original (b) nova posição

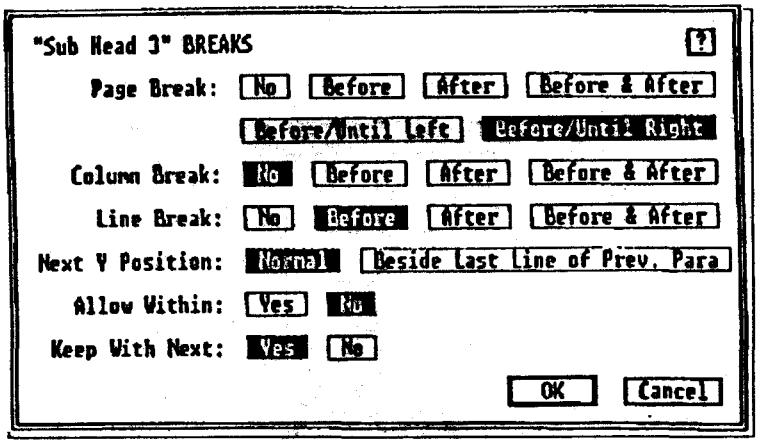

Figura 1.5: Tarefas representando um diálogo assíncrono

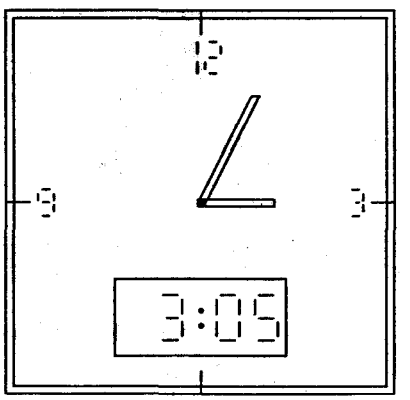

Figura 1.6: Exemplo de diálogo concorrente

e, através de **passos seqüenciais** modifica ou seleciona a opção desejada. Já no **mundo modelo**, o usuário **seleciona e executa diretamente** a opção desejada, representada na tela através de ícones, e utilizando para isto, geralmente, o *mouse*. Este tipo de diálogo, em que as opções são selecionadas diretamente de forma assíncrona, proporciona um uso mais intensivo do produto em função de uma operação mais direta por parte dos usuários.

Os dois tipos de diálogos são amplamente usados na construção de interfaces. A tendência, contudo, é o uso cada vez mais freqüente da filosofia assíncrona. O impacto desta forma de diálogo é enorme na construção de interfaces; inicialmente ela foi utilizada e implementada em computadores pessoais da linha Apple[14], onde foram introduzidos sistemas de múltiplas janelas com manipulação assíncrona na linha Macintosh. Hoje em dia esta filosofia já migrou para outras linhas e fabricantes, *e.g.* Sun Workstations, IBM-PCs, dentre outras.

1.4 Aspectos humanos

Para os projetistas de software, envolvidos com a construção de interfaces, os aspectos humanos, cada vez mais, são objetos de consideração. Pesquisas têm sido realizadas nas áreas de **ergonomia**, **psicologia cognitiva**, **psicologia comportamental**, **artes visuais** e **engenharia de fatores humanos**. Apesar de não ser este o ponto fundamental do presente trabalho, que envolve aspectos não só relacionados com a Ciência da Computação, fica registrado uma síntese dos pontos deste lado "mais humano" da construção de interfaces.

Segundo Feiler [Fei87], aspectos dos mais diversos podem ter influência na performance de um sistema: "Para alguns usuários, o uso de *menus* em sistemas pode ter uma usabilidade bastante precária, enquanto que para outros não". A afirmativa de que a primeira análise pode parecer insolúvel, do ponto de vista meramente técnico, tem fundamento; se um digitador, por exemplo, em seu ambiente de trabalho, está habituado apenas à entrada de dados de uma forma textual, seria muito desgastante para o mesmo ter que ficar constantemente interrompendo o seu trabalho e ficar selecionando opções de *menus*. Problemas como este nos levam a crer que cada vez mais aspectos humanos têm tido influência significantiva na construção de sistemas computacionais. A **ergonomia** é a ciência que estuda as relações entre os seres humanos e o seu ambiente de trabalho. Ela estuda aspectos físicos dos ambientes tais como o formato ideal para um teclado, quais as teclas que devem estar mais destacadas, o desenho ideal dos monitores gráficos, e o formato do *mouse*, dentre outros.

Além da ergonomia, estudos na área de **psicologia cognitiva** têm-se mostrado relevantes na construção de sistemas, uma vez que ela desenvolve modelos de persepção, cognição e reflexos. Estes estudos são importantes para prover uma interação mais rápida por parte do usuário em relação aos questionamentos do sistema. Ela discute, por exem-

[14] Apple é marca registrada da Apple Inc.

plo, qual deve ser a cor de uma mensagem importante do sistema para que o usuário tenha o reflexo necessário para observá-la e respondê-la.

As **artes visuais** contribuem no desenvolvimento de interfaces no sentido de desenvolver símbolos e objetos (ícones) que possam representar graficamente figuras já existentes no mundo real cujo fim é facilmente entendido pelo usuário final.

A **engenharia de fatores humanos** está bastante relacionada com a **ergonomia**. Ela provê metodologias que ajudam os projetistas na concepção de sistemas. A **engenharia de fatores humanos** se concentra nos **aspectos do usuário** da interação homem-máquina.

1.4.1 Evolução do processo de interação

Existem muitos fatores que contribuem para o comportamento do usuário e para a adaptação deste às interfaces:

- funcionabilidade da interface;
- facilidade de aprendizado;
- tempo para desenvolver uma determinada tarefa e
- qualidade da saída dos resultados.

A análise e a percepção destes fatores do processo de evolução da interação têm uma importância fundamental para a adequação do usuário a um sistema e, conseqüentemente, à produtividade deste em relação à Empresa. Pesquisas na área mostram que não só aspectos relacionados ao hardware do equipamento, como por exemplo o tamanho ideal dos caracteres na tela ou a velocidade de apresentação dos resultados, são importantes para um aumento de produtividade: "A produtividade aumenta sempre que o tempo de resposta do computador decai".

O número e a complexidade das variáveis para se construir boas interfaces parece estar cada vez mais em curva crescente.

1.4.2 Representações pictoriais

"A forma pela qual as informações são apresentadas influencia fortemente o tempo que a mente humana leva para processá-las". Com esta afirmação, segundo Feiler [Fei87], fica claro que os projetistas de software devem se preocupar, também, com aspectos relacionados à forma de apresentação dos resultados. A mente humana pode "digerir" os resultados apresentados de diferentes formas: se a apresentação for feita de forma gráfica, por exemplo, os seres humanos podem processar a informação apresentada de uma forma mais

global[15] para depois passar a uma análise mais detalhada do objeto. Se a apresentação for feita de forma textual, o processo de análise será necessariamente seqüencial.

As apresentações de forma gráfica provêm mais dimensões de observação e detalhes de percepção mais rápida por parte do usuário. As apresentações textuais possuem apenas uma dimensão de observação que é a cadeia de caracteres, a qual pode ter algumas modificações a fim de permitir um aumento de percepção por parte do usuário, *i.e.*, tipo de fonte adotado, caracteres itálicos, caracteres em negrito, etc. Já nas apresentações gráficas, as dimensões, que são de três, podem ser de certa forma estendidas se forem levados em consideração aspectos como cores, tipos de fontes, distância, escala, formato, dentre outras, que ajudam a mente humana a distingüir e digerir o objeto em análise.

Um outro fato é que nas apresentações gráficas os usuários lidam diretamente com "realidades" virtuais, enquanto que nas apresentações textuais eles se relacionam com nomes e símbolos que representam a visualização de um objeto, dificultando o processo de percepção desejado.

A partir destas afirmações, é possível dizer que no processo de projeto de um sistema, observações relativas à psicologia dos usuários devem ser consideradas, pois **"cada vez mais os sistemas destinam-se para usuários não especialistas"**. A evolução dos princípios de adequação do usuário a sistemas computacionais tornou-se significante a partir de relatos e opniões dos mesmos sobre o seu instrumento de trabalho.

1.5 Independência de diálogos

1.5.1 Breve histórico

O desenvolvimento de interfaces homem-máquina começou a se tornar crítico à medida em que aplicações foram se tornando cada vez mais interativas, pois parte significativa da construção de aplicações era dispendida na interface (e na sua re-escrita para contextos ligeiramente diferentes dos anteriores). Desta forma, cada vez mais surgia a necessidade de se construir um mecanismo de auxilio aos projetistas, no desenvolvimento de sistemas. Ou seja, um mecanismo que permitisse aos analistas uma modificação no projeto da interface sem que houvesse tantos problemas, *e.g.* recompilação de módulos, adaptação dos aplicativos já em operação às novas características da interface, etc.

Os pesquisadores envolvidos na área de **bases de dados** também passavam por problemas similares, pois a simples modificação de um atributo numa tabela, por exemplo, era suficiente para requerer uma reconfiguração completa do banco de dados envolvido e uma recompilação/adaptação de todos os aplicativos envolvidos. A solução para este problema veio a seguir com a implementação dos conceitos de **independência de dados** cujo foco principal era a separação entre uma base de dados e os programas a ela relacionados.

[15] no sentido de não seqüencial.

Um conceito similar foi introduzido para diminuir o trabalho dos engenheiros de software na construção de interfaces: **a independência de diálogos**. Este conceito baseia-se na idéia de que aspectos de interação **apenas** relacionados com o diálogo homem-máquina poderiam ser **isolados** dos aspectos funcionais específicos de aplicativos e programas de um sistema. Na prática, isto significa que a estruturação da apresentação de *menus*, comandos do sistema, entrada/saída de dados, diálogos do tipo pergunta/resposta, ou seja, todas decisões que se referem apenas à interface ou ao usuário final são ortogonais à estruturação de um sistema computacional.

1.5.2 Motivação para a independência de diálogos

A descrição do exemplo a seguir servirá como base para a elucidação da motivação para se construir diálogos independentes de programas de sistema computacional.

Uma Empresa hipotética não adotou a independência de diálogo como objetivo do processo de desenvolvimento do software em seu departamento de informática. Neste caso a estrutura de dados do sistema e toda a parte relacionada com a interação com o usuário estava incorporada aos programas do sistema. Desta forma, qualquer alteração em algum detalhe da interface modificaria, inevitavelmente, os programas específicos da aplicação. A simples mudança do número de opções de um *menu*, ou até mesmo a mudança do tipo de diálogo provido por um *menu* do sistema, resultaria na re-escrita e testes de partes significantes de **diversos** programas que fizessem menção à nova situação. A primeira grande vantagem, então, do desenvolvimento de diálogos independente dos aplicativos específicos do sistema é a **flexibilidade** do projeto da interface.

Uma mudança como a descrita acima, em sistemas, cujo desenvolvimento foi realizado sob a filosofia da independência de diálogos, seria "permissível" e menos custosa, pois esta mudança seria realizada[16] no módulo que tratasse desta interação, sem que houvesse modificações nos programas específicos[17].

Uma outra propriedade e vantagem da utilização da independência de diálogos é a **consistência** de propriedades. A utilização de um módulo independente, por exemplo, para a implementação da troca de mensagens entre o usuário e a máquina, evitaria a diversidade de alternativas para a efetivação deste diálogo. Este fato poderia ser ilustrado como a seguir: "no modo de alteração de cadastros, utilize a tecla **PF1** para invocar a função *help*; no modo de consulta de cadastros utilize a tecla **H** para o *help*; para a utilização do auxílio no modo de impressão, utilize a tecla **<ESC>**".

Este exemplo ilustra o fato de cada programa possuir o seu próprio **domínio de diálogos**, o que daria, na junção das funções, a diversidade de opções para efetivação do

[16] na realidade será visto que a escrita de código abstrato e o uso de programação orientada a objetos eliminaria grande parte (ou toda) das mudanças em diálogos de entrada/saída.

[17] a incorporação de uma biblioteca de classes interativas, *i.e.* *menus pulldown*, *menus popup*, *menus dialog* em sistemas concebidos com independência de diálogos seria também válida para viabilizar mudanças em tipos de *menus* ou em tipos de interações.

diálogo, como ilustrado.

A independência de diálogos incorpora, desta forma, inúmeras vantagens sobre a sua não utilização: **consistência, flexibilidade, confiabilidade** (execução em diferentes plataformas, acoplamento a uma aplicação específica, etc.), **clareza de código, extensibilidade** (acréscimo de novas partes sem que o sistema pré-existente necessite ser interrompido), **independência lógica** (separação entre os aspectos sintáticos e os aspectos semânticos, *i.e.* a interface e a aplicação), dentre outros.

1.5.3 Problemas relacionados à independência de diálogos

Apesar das inúmeras vantagens que a adoção da filosofia de independência de diálogo pode trazer, alguns problemas são inerentes quando da efetivação desta mesma independência. Geralmente, boas técnicas de programação são, inevitavelmente, confundidas com a independência; afirmações do tipo: "As rotinas de erros foram implementadas num programa à parte do sistema; ou ainda, toda a parte de tratamento da entrada e saída do usuário é implementada num procedimento especial", foram amplamente utilizadas (e confundidas) para "representar" a independência de diálogos. Na verdade, os programadores tentam fazer com que **boas técnicas de programação se aproximem dessa independência.**

"A independência de diálogos deve ser um mecanismo presente durante toda a fase de concepção e implementação de um sistema computacional". Uma aplicação interativa, desta forma, deve ser composta, para sistemas desenvolvidos sob esta filosofia, de dois componentes: o **componente de diálogo** que é o módulo onde se efetua a comunicação entre o usuário final e o sistema, e o **componente computacional**, que é o módulo onde estão situadas as rotinas específicas da aplicação e onde, em geral, não há interação com o usuário.

Estes dois componentes devem estar inteiramente separados durante todas as fases de concepção de um sistema: desenho, projeto, manutenção, etc. Entretanto, algumas dificuldades são inerentes à adoção destas técnicas:

- a separação do código gerado entre os componentes **computacional** e de **diálogo** pode afetar a performance do sistema, uma vez que a comunicação entre os elementos destes componentes se dará, em sua maioria, em tempo de execução;
- como o desenvolvimento dos dois componentes se dá em plataformas distintas, uma maior interação e comunicação entre os projetistas do diálogo e os implementadores do sistema terá que ocorrer;
- apesar de o código relacionado ao diálogo e ao estilo de interação (uso do *mouse*, *menus*) estar separado do **componente computacional**, uma grande quantidade de código fonte terá que ser escrito (e re-escrito) quando do surgimento (e alterações) de novos estilos de interação, com o usuário[18].

[18] este é um problema que, inevitavelmente, os adeptos ou não da independência de diálogos, terão que

As dificuldades podem se agravar se o diálogo assíncrono (seção 1.3) for adotado para a construção de interfaces. Por outro lado, no diálogo seqüencial, a entrada do usuário já tem um "caminho" ou uma "árvore de caminhos" mais ou menos pré-definida. Desta forma, a separação entre os dois componentes pode se dar sem que ocorram tantos problemas como no diálogo assíncrono. O usuário percorre uma rede de *menus* e vai realizando a sua interação, culminando com a efetivação de uma ação submetida ao sistema (*i.e.* a ativação de uma opção, a atribuição a um "flag" do sistema, etc.). Já no diálogo assíncrono, o usuário desenvolve as operações diretamente com os componentes do diálogo, e de uma forma "multi-tarefada". A separação se dá de uma maneira mais complexa pois o **componente do diálogo está entrelaçado** com o **componente computacional**. O usuário pode, em um mesmo nível, disparar processos, selecionar opções e ativar *flags*. Ou seja, os elementos dos dois componentes têm comportamento que são **reflexos** de uma **estrutura de dados comum** às duas entidades (interface e aplicação).

Mesmo com todas as dificuldades para se implementar a independência de diálogos, esta separação está cada vez mais presente em projeto de sistemas, devido, principalmente, à utilização de novas metodologias e técnicas de desenvolvimento de sistemas tais como:

- a descoberta de novas arquiteturas que implementam o processamento concorrente dos componentes de **diálogo** e **computacional** absorve o problema da queda de performance decorrente da separação;
- a utilização de novas metodologias de projeto de sistemas para melhorar a fase de comunicação entre os projetistas da interface e os implementadores da aplicação;
- a solidificação das técnicas de escrita de **códigos abstratos** e de **programação orientada a objetos** nos procedimentos de interação com o usuário,

foram significantivos para a efetivação desta separação.

1.6 Representação de interfaces homem-máquina

Os projetistas de aplicações necessitam de notações para representação dos diálogos. Diversas destas têm sido utilizadas para a **representação de interfaces entre o homem e a máquina**. No início, estes mecanismos se resumiam a representações textuais, que depois evoluiram até as representações gráficas e, mais recentemente, aos sistemas de representação de diálogos, ferramentas que auxiliam os projetistas a desenvolver representações de diálogos, gráficas e/ou textuais, de maneira automática.

enfrentar. O surgimento de novas tecnologias de entrada/saída, associadas a novos estilos de interação tem influência significante na re-escrita de código no **componente de diálogo**.

1.6.1 Metalinguagens (BNF, ...)

A modelagem da interação do usuário com a aplicação tem sido baseada na adoção de **metalinguagens**: linguagens para representar outras linguagens. Entretanto, a sua utilização sistemática tem apresentado uma série de problemas. Elas geralmente envolvem notações e símbolos, de difícil compreensão e assimilação. Diversas variantes de metalinguagens têm sido utilizadas para representar diálogos homem-máquina: BNF, expressões regulares, gramáticas livres de contexto, diagramas de transição de estados, redes de Petri, etc. O grande atrativo dessas variantes é a existência de notações gráficas que facilitam a construção de ferramentas para apoiar a própria tarefa de representação. Por outro lado, elas não têm se mostrado poderosas na representação de todos os nuances relacionados com a **representação de um diálogo** pois, assim como os formalismos adotados para representar linguagens de programação, as metalinguagens não conseguem expressar, com rigor, **aspectos semânticos** da interação[19]. Desta forma, variações destes formalismos foram criados com o fim de refletir estes aspectos (*i.e.* RAPID/USE [HH89]). Qualquer que seja o formalismo adotado, este deve capturar e permitir expressar adequadamente os conceitos de estado, ações do usuário, validação por parte do sistema, ações de resposta da aplicação e transição entre estados da interação.

As metalinguagens não conseguem explicar o processo de como, por exemplo, um comando é solicitado pelo sistema (*e.g.* exibindo um *menu* de ícones) e respondido pelo usuário (*e.g.* selecionando um ícone). As técnicas adicionais utilizadas para esta representação "tentam" contornar este problema. No sistema RAPID/USE [HH89] (uma variante de diagramas de transição de estados), por exemplo, mecanismos de entrada de dados e solicitações do sistema (*i.e.* exibição de *menus*) são descritos dentro de cada nódulo do diagrama através de linguagens de programação de diálogos **textuais** (subseção 1.6.2) e, os arcos entre os nódulos do diagrama são usados para representar, por exemplo, os diversos eventos e alternativas (*i.e.* as opções de um *menu*) que o nódulo em questão pode assumir[20].

1.6.2 Representação de diálogo seqüencial

Representação BNF

Uma das melhores formas de representação de sintaxe de linguagens é, sem dúvida, a Backus Normal Form (BNF). Entretanto, quando utilizada para representar diálogos, apresenta algumas deficiências. Em particular, quando tenta representar aspectos de **contexto** de diálogos. O BNF é uma representação textual, e por isto detém uma visualização de compreensão e assimilação bastante difícil a uma primeira análise. Esta forma de repre-

[19] em descrições de linguagens de programação como em Kowaltowski [Kow83], a BNF foi adotada para a representação apenas de **aspectos sintáticos** da linguagem. O formalismo BNF puro não consegue captar os aspectos semânticos das linguagens.

[20] ver figura na seção 1.6.2

<programa> ::= program <identificador>;
<bloco>.

...

<lista de expressões> ::= <expressão>
{,<lista de expressões>}

<expressão> ::= <expressão simples>
[<relação>] <expressão simples>

<termo> ::= <fator> {(* | div | and)} <fator>

...

Figura 1.7: Exemplo de uma representação BNF (extraída de [Kow83])

sentação baseia-se em símbolos **não-terminais** e símbolos **terminais**, conforme ilustra a figura 1.7[21]. Os símbolos **não-terminais** podem ser substituídos por produções de elementos **não-terminais** que irão produzir mais elementos **não-terminais** e/ou **terminais**. O processo pode se repetir inúmeras vezes até que só existam símbolos terminais. Todo este processo faz com que seja muito difícil visualizar sentenças numa linguagem, apenas a partir de sua descrição BNF.

A representação por BNF tem sido usada, também, para a representação de diálogos. Alguns sistemas para simulação foram desenvolvidos para aceitar uma descrição textual BNF (associadas com regras para descrever ações) e produzir protótipos de interação homem-máquina.

Os símbolos da metalinguagem BNF, contudo, não promovem uma organização estrutural do diálogo. A BNF é **uma notação sintática que pode representar instâncias específicas de um diálogo.**

Representação por diagrama de transição de estados

Os diagramas de transição de estados podem ser considerados como sendo uma segunda forma de representação de diálogos seqüênciais. Apesar de, originalmente, os diagramas de transição de estados terem sido idealizados para a representação de linguagens (como o BNF), adaptações e variantes são utilizadas para a representação de diálogos.

[21] os símbolos não-terminais estão representados por letras em estilo negrito e os terminais por letras em estilo normal.

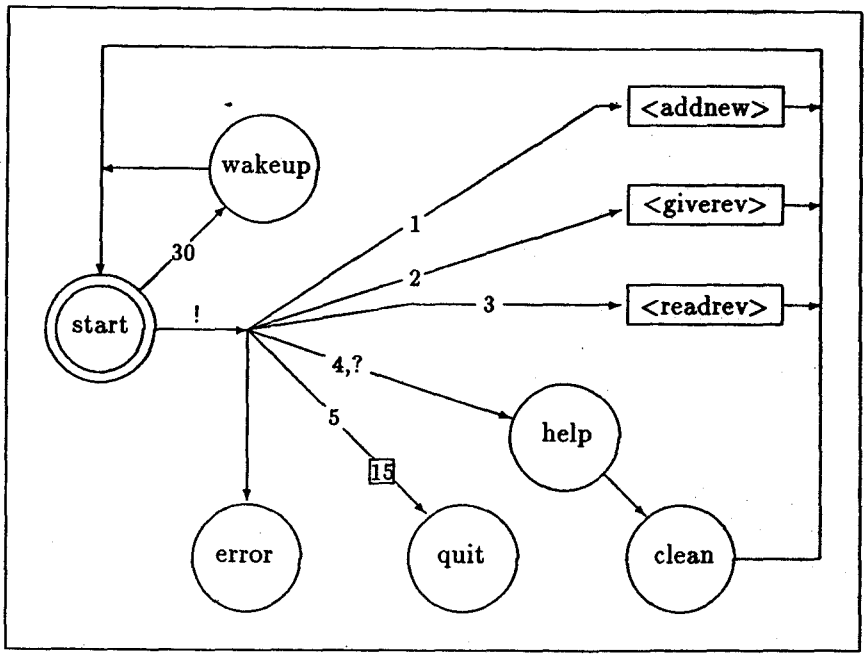

Figura 1.8: Exemplo de um diagrama de transição de estados

Enquanto a representação BFN constitui-se de um formalismo que "representa" diálogos de forma textual, a representação por diagramas de transição de estados os simboliza de forma gráfica através de **nódulos** e **arcos**. Os nódulos representam os **estados** do diálogo enquanto que os arcos representam as **transições** entre os estados.

A figura 1.8 ilustra o exemplo de um diagrama de transição de estados de um diálogo adaptado de Hartson [HH89]. O **estado inicial** é representado por duas circunferências circuncêntricas (o estado `start`), os números entre os arcos denotam os eventos que devem acontecer para que a mudança de estado ou **transição de estados** possa ocorrer; e os retângulos representam procedimentos (ou outros módulos) que são executados quando da ocorrência do evento associado.

A utilização de diagramas de transição de estados na representação de diálogos sofreu algumas modificações na sua forma original com o objetivo de representar nuances da interação homem-máquina, como por exemplo, opções de *menus*, ações a serem executadas na escolha de opções, exibição de mensagens de *help*, etc.

O diagrama da figura 1.8, por exemplo, representa um *menu* com cinco opções: <addnew>, <giverev>, <readrev>, *help* e *quit*; o tratamento dos erros (entradas inválidas, neste exemplo) também são representados nestes diagramas. Na figura 1.8, a seleção de uma opção inválida faz com que o ocorra uma trasição no diagrama para um novo estado

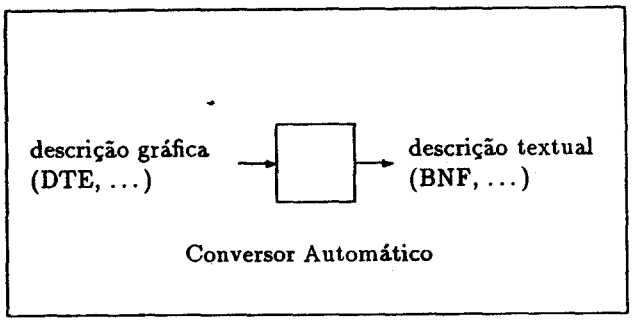

Figura 1.9: Esquema da funcionabilidade de conversores gráficos → textuais

(error), onde o tratamento será executado.

Alguns aplicativos foram utilizados, a partir da representação por diagramas de transição, para se chegar a representações textuais correspondentes. Estes programas recebem como entrada um diagrama de transição de estados e devolvem a descrição textual (que pode ser inclusive, uma descrição BNF) da interface representada. A figura 1.9 ilustra, de uma maneira esquemática, a funcionalidade de aplicações como esta.

A figura 1.10 ilustra a representação textual do diagrama de transição de estados exibido na figura 1.8. A adaptação é de um exemplo apresentado por Hartson [HH89], e apenas parte da descrição foi representada.

Representação por linguagem de diálogo

Os diálogos também podem ser expressados por linguagens de programação. Qualquer que seja a linguagem escolhida, ela deve permitir construções que possam representar as propriedades de um diálogo; tais como: o fonte a ser adotado na exibição de um *menu*, os atributos do vídeo, posicionamento de cursor e mensagens, cores, dentre outros.

Diversas linguagens para representação de diálogos foram concebidas e têm sido utilizadas. A linguagem de diálogo ilustrada para o exemplo da figura 1.8 foi mostrada na figura 1.10. Neste caso particular, a tela de um terminal alfanumérico é montada linha por linha, como ilustrado, e alguns caracteres de controle foram introduzidos para representar algumas características de posicionamento e atributos de exibição. Na figura 1.11 são exibidas duas linhas da linguagem de diálogos exemplificada na figura 1.10. A primeira linha significa que na segunda linha do terminal, depois da linha corrente e, na coluna 10, a mensagem: "1: Add new restaurant to database" será exibida. Na segunda construção a mensagem "Press RETURN to continue" será exibida na penúltima linha do vídeo, em modo reverso (*reverse video - rv*) e centralizada em 80 colunas. Depois da exibição, o atributo do vídeo retorna para o normal (*standard video - sv*).

```
diagram irg entry start exit quit

node start
    cs, r2, rv, c_'Interactive Restaurant Guide',sv,
    r6,c5, 'Please make a choice:',
    r+2, c10, '1:  Add new restaurant to database',
    ...

node help
    cs, r5, c0,'This program stores and show ...',
    ...

node error
    r$-1, rv, 'Illegall command', sv, 'Please
              type a number from 1 to 5.',
    ...

arc start single_key
    on '1' to <addnew>
    on '2' to <giverev>
    ...
    on '5' to quit
    alarm 30 to wakeup

...
```

Figura 1.10: Descrição textual da representação por diagramas de transição de estados da figura 1.8

```
r+2, c10,
   '1:  Add new restaurant to database'

...

r$-1, rv, c80, c_
   'Press RETURN to continue', sv
```

Figura 1.11: Exemplo de uma linguagem de diálogos

Outras funções como, por exemplo, entrada e validação de dados, facilidade de edição de campos, etc., são também utilizadas na construção de linguagens de representação de diálogos.

1.6.3 Representação de diálogo assíncrono

Representação baseada em eventos

A representação baseada em eventos constitui um dos mecanismos mais poderosos para a representação de diálogos assíncronos. Diversas técnicas para a representação de diálogos assíncronos são variantes do mecanismo da representação baseada em eventos.

No SGIU da Universidade de Alberta [HH89], uma variante da representação controlada por eventos, a construção de interfaces homem-máquina é estabelecida através da *event handlers*, que são declarações de eventos descritos através de uma "linguagem de eventos" muito similares à sintaxe da linguagem de programação C [KR78]. A declaração de um diálogo descrito através de uma linguagem de eventos consiste de pelo menos uma declaração de eventos como ilustrado na figura 1.12. A declaração é dividida em três partes:

1. entradas e saídas - declaração dos átomos[22] que o gerenciador de eventos irá processar;
2. variáveis locais - usadas para a finalização das ações a serem implementadas no *event handler*;
3. declarações dos eventos - comandos da linguagem de programação C que implementam as ações dos eventos.

A parte visual da interface é concebida e representada através de um pacote gráfico, baseado em janelas, e incorporado ao projeto.

Uma característica deste SGIU é a sua similaridade com a linguagem C (o que o torna, de certa forma, limitado aos programadores C).

Outros mecanismos de representação de diálogos baseada em eventos são atualmente usados, como por exemplo o sistema ALGAE (A Language for Generating Asynchronous Event handlers) e o sistema ERS (Event-Response System). Em ambos os mecanismos, os eventos são descritos através de uma linguagem de programação. No sistema ALGAE, um evento é uma estrutura de dados que possui um **tipo** e um **valor** e a sua especificação é usada na geração de *event handlers*, uma estrutura utilizada para o gerenciamento da entrada de dados do usuário. O sistema ERS, por sua vez, foi concebido especificamente para a representação de diálogos "multi-tarefados". Os seus eventos são descritos e especificados através da Event-Response Language (ERL), uma linguagem baseada na representação de respostas dos sistema resultantes das ações por parte do usuário final.

[22] uma seqüência de caracteres que representa um símbolo terminal da linguagem (*token* - em inglês).

```
Eventhandler event_handler_name Is
  Token
    token_name event_name;
    ...
  Vars
    type variable_name = initial_value;
    ...
  Event event_name :  type {
    statements
  }
  Event event_name :  type {
    statements
  }
end event_handler_name;
```

Figura 1.12: Exemplo de um SGIU baseado em eventos

1.7 Ferramentas interativas para o desenvolvimento de interfaces

1.7.1 Introdução: software *toolbox*, *toolkit* e SGIU

O desenvolvimento de interfaces homem-máquina através de ferramentas tornou-se imprescindível para os sistemas atuais, pois estes sistemas estão cada vez mais interativos e com suas interfaces mais elaboradas. Desta forma, o desenvolvimento de ferramentas e aplicativos que auxiliem projetistas no desenvolvimento de sistemas tornou-se um campo de pesquisa e produção crescentes.

As ferramentas para o desenvolvimento de interfaces, além de usadas para a implementação das interfaces propriamente ditas, também são utilizadas para a sua representação (produção de diagramas de transição de estados, BNF's, ...). As fases de utilização destas ferramentas incluem as etapas de criação de protótipos, avaliação, análise, implementação e testes. Desta forma, estas ferramentas devem incluir editores de texto, editores gráficos, sistemas de gerenciamento de bancos de dados (para produção de descrições de tabelas, por exemplo), editores de diagramas de transição de estados, etc.

Como em toda área emergente, a terminologia básica não está, ainda, completamente consistente. O termo ferramenta (*tool*), por exemplo, tem sido usado neste texto (e nas referências) para denotar desde simples rotinas interativas até um ambiente completo para desenvolvimento de interfaces.

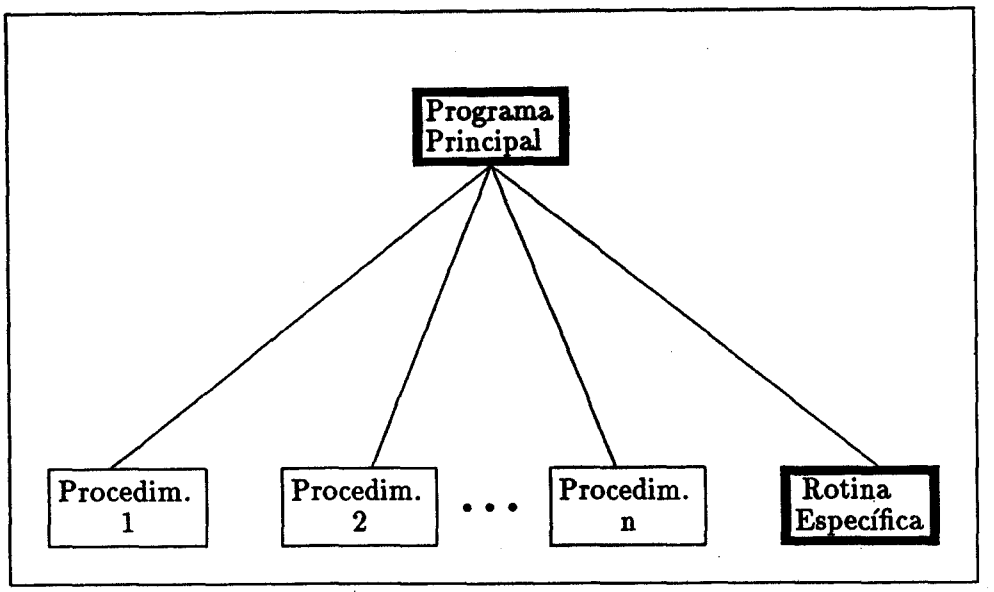

Figura 1.13: Estrutura geral de software desenvolvidos sob a filosofia *toolbox*

As aplicações, entretanto, podem sofrer uma espécie de **rotulação** quanto à filosofia adotada para a sua construção:

- filosofia *toolbox*[23] - são pacotes em que o projetista da aplicação constrói o corpo principal do programa e as rotinas específicas da aplicação. Nos pontos do programa onde existir necessidade de interação com o usuário, seja através de *menus* ou de uma entrada de dados, o projetista incluirá chamadas às funções[24] já projetadas para este fim. Esta forma de desenvolver software faz com que as aplicações possam ser prototipadas, de uma maneira bastante rápida comparadas às técnicas tradicionais. Em compensação, alguns problemas são inerentes a esta técnica como, por exemplo, a impossibilidade de o projetista poder fazer ligeiras modificações nas rotinas, para melhor adequá-las à sua aplicação.

  Na figura 1.13 é ilustrada estrutura geral de uma aplicação desenvolvida sob a filosofia *toolbox*.

  Exemplo: Sistema GKS (Graphical Kernel System) [Har87], o X WINDOW System [Jon89].

[23] no sentido literal da palavra, ou seja, uma caixa com peças de construção de interfaces em que o seu uso e escolha estão a cargo do projetista.

[24] o termo funções inclui funções e procedimentos.

- filosofia *toolkit* (ou software *framework*) - Este tipo de aplicação difere da abordagem *toolbox*, no sentido que **aplicações virtuais** podem ser construídas, ao passo que na abordagem *toolbox* as rotinas de interface já estão prontas para serem usadas, sendo constituídas de funções com código relocável e imutável[25]; na abordagem *toolkit* existem as funções virtuais que permitem que o seu **código virtual** correspondente possa ser especializado. Aqui, o projetista pode "ajustar" *menus*, modificar formato de janelas, etc, podendo assim, especializar uma coletânea de **classes** pré-existentes e adequá-las ao "seu jeito"

  Na figura 1.14 é ilustrada a estrutura geral de aplicações desenvolvidas sob a filosofia *toolkit*.

  Exemplos: ET++ [W+88], Interviews [LCV87], MACAPP [Sch86a], XView Toolkit [Hel90].

- Sistemas de Gerência de Interface com Usuário (SGIU)[26] - podem ser considerados como um conjunto de programas de alto nível, que permitem o projeto, a criação de protótipos, execução, evolução e manutenção de interfaces e ainda com a propriedade de que todos esses programas estejam sobre um simples gerenciador de interfaces. O uso de sistemas SGIU implica em dizer que a construção da aplicação está isolada de detalhes de implementação da interface. O SGIU gera automaticamente aplicações interativas, de acordo com a especificação fornecida pelo projetista. Esta especificação pode ser representada por linguagens de programação, gramáticas (BNF's-*like*, diagramas de transição de estados, ...(seção 1.6).

  Exemplos: O Alberta UIMS [HH89], o MENULAY UIMS [HH89] (um sistema de alto nível que possui uma espécie de pré-processador que "traduz" *menus* e gráficos para programas em C os quais quando executados, produzem os *menus* e gráficos especificados na descrição da interface), o MOUSE UIMS [Cou87], o SYNGRAPH UIMS (SYNtax directed GRAPHics) [HH89] e o MIKE (Menu Interaction Kontrol Environment) [HH89].

1.7.2 Características das ferramentas de desenvolvimento de interfaces

As ferramentas utilizadas na construção de interfaces devem ter algumas características básicas:

- **Funcionalidade** - no sentido de "o que a ferramenta pode produzir e fazer": que estilos de interação ela suporta, que técnicas são utilizadas para produzir interfaces, que dispositivos de entrada e saída podem ser usados nas interfaces, etc. Esta característica é fundamental, pois as aplicações interativas atuais são complexas,

[25] no sentido de não especializável.

[26] do Inglês User Interface Manegement Systems - UIMS.

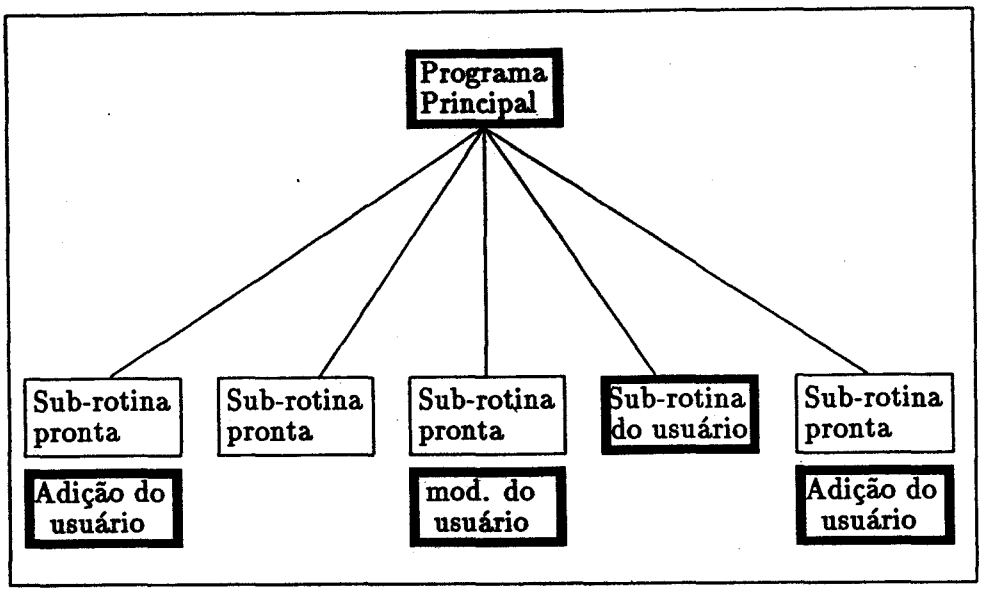

Figura 1.14: Estrutura geral de software desesenvolvido sob a filosofia *toolkit*

manipulam diversos dispositivos de entrada/saída e suportam diferentes técnicas de gerenciamento de diálogos.

- **Usabilidade** - esta também é uma característica fundamental destas ferramentas, pois diz respeito à satisfação e facilidade do uso (por parte dos projetistas) destes programas. Assim como uma das principais características de sistemas produzidos para o usuário final é a facilidade e simplicidade no aprendizado, as ferramentas utilizadas pelos projetistas (apesar de serem usadas para a construção de programas complexos) também deveriam ser de simples manuseio, para garantir sua efetiva utilização por parte dos projetistas.

- **Portabilidade** - as ferramentas devem ser o menos possível dependentes de máquinas e compiladores, a fim de permitir a **portabilidade** (execução em diferentes plataformas) do aplicativo. Além disto, a portabilidade diz respeito à facilidade de adaptação de alguns procedimentos de baixo nível (*e.g.* exibição de um pixel na tela) nos diferentes ambientes a serem executados.

- **Extensibilidade** - diz respeito à facilidade de incorporação de novas técnicas de interação ao aplicativo. Existem pelo menos dois caminhos que fazem um aplicativo gerador de interfaces extensível: ou ele possui facilidades de adaptação a novas condições (novos dispositivos, novos estilos de interação, etc.) ou a própria repre-

sentação da interface por ele gerada possa ser facilmente modificada.

- **Manipulação direta** - assim como para os usuários finais, o desenvolvimento de ferramentas com técnicas de manipulação direta é desejável. O projetista deve poder trabalhar diretamente através de editores gráficos, para poder confeccionar representações gráficas de diálogos, *menus*, etc.

- **Integração** - diz respeito à necessidade de se ter uma única interface para possibilitar o acesso a todas as ferramentas do aplicativo. Uma interface integrada e um estilo de interação uniforme são essenciais no uso destas ferramentas.

- **Consistência** - está relacionada com a uniformidade dos objetos interativos manipulados na interface, ou seja, a necessidade de definição de uma estrutura de dados única para definições que se aplicam a várias partes de um sistema. Um exemplo desta propriedade seria a apresentação de *menus* com a mesma cor, a mesma posição para o título, o mesmo tipo de fonte para os textos exibidos, etc. A mudança de um item desta estrutura (a cor, por exemplo) resultaria na propagação da modificação em todas as aparições de *menus* a ela relacionadas.

Capítulo 2

Programação Orientada a Objetos

2.1 Introdução

2.1.1 Motivação

Metodologias de desenvolvimento de aplicações têm sofrido modificações, ao longo dos últimos anos. Estas evoluções têm sido observadas, por exemplo, na arquitetura interna que o sistema deve possuir, uma vez que o tamanho de um software pode chegar a dezenas de megabytes de código fonte. Modificações neste setor incluem a exploração de paralelismo, a reutilização de código e a implementação de rotinas com forte interação gráfica, dentre outras. Além de evoluções no campo da arquitetura interna dos sistemas, as aplicações têm evoluído visando assistir usuários não especializados na execução de tarefas relevantes. Estas evoluções foram induzidas pela necessidade de os sistemas se tornarem cada vez mais confiáveis. São notadas, por exemplo, modificações na forma de interação com o usuário, conforme dito no capítulo 1, e também em mecanismos de proteção contra erros e omissões por parte dos operadores do sistema (sistemas de controle de tráfico em vias férreas e aéreas, por exemplo, devem estar habilitados para "resolver" certas situações de conflito sem que seja necessário a intervenção do operador). Junte-se a estas novas características o fato de os sistemas atuais serem cada vez mais complexos (execução de dezenas de milhares de cálculos, coordenação entre módulos, etc.), voláteis[1] e distribuídos.

Um outro aspecto importante neste processo é o de saber qual deve ser a ênfase e qual a orientação a ser adotada (orientação a dados, orientação a processos, **orientação a objetos**) no desenvolvimento de um sistema. Uma abordagem emergente, que leva em consideração os problemas e as características de aplicações atuais como os descritos acima, é a abordagem baseada no **paradigma de objetos**. Vemos, na utilização desta metodologia, uma preocupação em resolver (ou pelo menos amenizar) problemas

[1] a palavra volátil, no contexto de desenvolvimento de software, está relacionada com as constantes modificações sofridas pelos programas no seu desenvolvimento.

relacionados, por exemplo, com a volatilidade de software, a independência de módulos (e maior confiabilidade do produto final) bem como a estruturação da arquitetura interna do sistema. A preocupação na solução do problema se concentra através da adoção de mecanismos que permitem a ligação de tipos e módulos em tempo de execução de programa, da implementação dos conceitos de **tipos abstratos de dados**, da adoção da reutilização de código, e da estruturação de módulos utilizando os conceitos de **herança** de tipos e operações, dentre outros.

Não só as metodologias baseadas no paradigma de objetos, mas qualquer metodologia emergente deve levar em consideração três aspectos essenciais na construção de software: que ele possa ser implementado de uma maneira **rápida, barata** e **flexível**. As linhas de ação para enfrentar estes problemas abrangem, inevitavelmente, a descoberta de novas técnicas para desenvolvimento de software e a reestruturação dos métodos de implementação de sistemas.

A adoção do paradigma de objetos no desenvolvimento deste trabalho se dá em função de haver um sentimento emergente de que esta linha de desenvolvimento **aglutina** propostas de soluções para enfrentar alguns dos problemas citados. Soluções para se construir software rápido, barato e flexível têm sido logradas através de metodologias de desenvolvimento para a orientação a tipos abstratos de dados e da criação de módulos independentes (construir rápido); da utilização e implementação de reutilização de código e uma conseqüente diminuição de tempo de desenvolvimento (construir barato) e no suporte à volatilidade de software com a adoção de conceitos de **acoplamento dinâmico**[2], que viabiliza a "fuga do rígido e inflexível" (construir flexível).

2.1.2 Paradigmas de programação

O termo paradigma, de uma forma geral, foi utilizado na década de 1970, para fazer referência a um conjunto de crenças e preconceitos que permearia a atividade dos membros de um corpo científico em uma certa época. Com o passar dos anos, o termo foi se "popularizando" e, já no final da década de 1970, o termo começou a ser utilizado no universo da programação de computadores para denotar **estilos de programação** e **mecanismos lingüísticos** que suportam tais características.

O termo paradigma de programação se popularizou tendo em vista, principalmente, a ênfase que foi dispendida, no final da década de 1970 e no início da década de 1980, à experimentação de novas linguagens e técnicas associadas de programação.

O primeiro paradigma de programação foi o **procedural** e que pode ser resumido com a seguinte afirmação de Stroustrup [Str88]: "decida que procedimentos você necessita e implemente-os usando os melhores algoritmos que você possa encontrar". Esta afirmativa sugere que o centro de todo o paradigma procedural é a definição **de que (e quais) procedimentos** serão necessários para realizar uma determinada tarefa. Algumas linguagens de programação associadas a este paradigma são: Fortran, Algol, C e Pascal.

[2]da expressão inglesa *late binding*.

Outros estilos sucederam e se entrelaçaram com o paradigma procedural. No final da década de 1970, apareceu o **paradigma funcional** que sugeria que as tarefas deveriam ser organizadas em termos de **funções**. LISP e KRC são exemplos de linguagens de programação que suportam o estilo funcional. Na década de 1980 surgiu o **paradigma lógico** e junto com ele linguagens como PROLOG.

Já no final da década de 1980, o termo paradigma se tornou tão comum a ponto de significar qualquer modificação na maneira de se programar. Vieram, então, os estilos **orientado por acessos, orientados a processos, orientados a objetos**, dentre outros.

Os paradigmas de programação têm surgido como **abstrações** de experiências com linguagens e sistemas. Isto nos leva a afirmar que os paradigmas não são necessariamente **ortogonais** entre si[3]. Uma constatação deste fato é o surgimento de linguagens de programação associadas a um determinado paradigma mas originárias de outros. é o caso de C++, espécie de linguagem **híbrida**, combinando o estilo de programação procedural da linguagem C com características do paradigma de objetos. Um outro caso é a linguagem CLOS (Common LISP Object System) derivada da linguagem funcional LISP, visando suportar conceitos associados ao paradigma de objetos.

2.1.3 Características das aplicações (e dos ambientes)

A forma de se desenvolver software vem sofrendo hoje em dia inúmeras modificações, conforme citado. No início, era bastante comum rotularem-se as aplicações em computadores, em função da linguagem e do equipamento utilizado. Hoje em dia, tal categorização não existe mais, e as aplicações (e seus ambientes) contemporâneas são identificadas pelo tipo de requisito que podem e devem atender:

- pela sua portabilidade ou capacidade de execução em diferentes plataformas e máquinas diferentes;
- pelo alto grau de paralelismo entre os seus módulos;
- pela execução e simulação de ambientes em tempo real;
- pelo elevado **grau de interação** com os usuários, apresentando todos os módulos através de uma interface gráfica icônica consistente (poder realizar a interação através de figuras e símbolos (ícones) - e não apenas textos - que representam uma adaptação de figuras existentes no mundo real e a sua representação em forma gráfica);
- pelo desenvolvimento do software **integrado** com ferramentas que auxiliam na tarefa de confecção (consistência de desenvolvimento);

[3]por exemplo, existem estilos de programação (e linguagens) que suportam tanto o paradigma procedural quanto o paradigma orientado a objetos.

- pelo suporte ao trabalho cooperativo (no sentido de particionamento de tarefas e realização de atividades em grupo).

2.2 Modelagem Conceitual

2.2.1 Domínio da aplicação

A construção de uma aplicação em informática, qualquer que seja a natureza, envolve um processo de mapeamento ou **representação** de uma atividade que está sendo realizada no mundo real num ambiente computacional. O desenvolvimento de programas em computadores pode ser resumido como: a representação de um **modelo** de dados em um espaço físico delimitado (que pode ser chamado de **espaço de soluções**) de um **problema** do mundo real (que é representado pelo **espaço de problemas**). O problema agora passa a ser como será este mapeamento físico ou como será esta representação no espaço de soluções.

Existe, desta forma, um espaço **abstrato** entre a realidade do mundo e a representação desta realidade no computador. A esta "distância" **abstrata** entre o espaço de problemas e o espaço de soluções dá-se o nome de *"gap"* semântico. E, claramente, quanto mais próximo estiverem estes dois espaços, mais fácil será o desenvolvimento de uma aplicação e, conseqüentemente, mais fácil será assegurar a corretude do problema, a confiabilidade e manutenção do software. O desenvolvimento de linguagens e ferramentas de quarta geração, cuja essência de projeto é fazer com que cada vez mais as definições do espaço de soluções sejam semelhantes ao fenômenos observados no espaço de de problemas, deixa evidente a preocupação que os pesquisadores têm em "eliminar" (conceitualmente) este *"gap"*.

A figura 2.1 representa e ilustra o *"gap"* semântico e as suas duas faces: o domínio de problemas e o domínio de soluções.

Feita então esta **representação abstrata** no espaço de soluções, cujas **operações** irão representar a simulação de operações sendo realizadas no mundo real, o próximo passo será a criação, pelo projetista da aplicação, de **algoritmos** que possam ser executados em computador e que implementem estas operações.

A identificação de que (e quais) pontos do problema do mundo real devem ser representados constitui-se na primeira grande tarefa do desenvolvimento de um software. A esta identificação das operações abstratas sobre um problema real é que os projetistas de software chamam de **análise do domínio da aplicação**, e que corresponde à modelagem das entidades e fenômenos da aplicação que o projetista julga relevante para o desenvolvimento do problema. Esta tarefa é conhecida por **modelagem conceitual** e sua efetivação leva em consideração a identificação das **abstrações** ou as operações mentais que devem ser executadas para capturar no domínio da aplicação a **representação** desta abstração em um modelo físico delimitado.

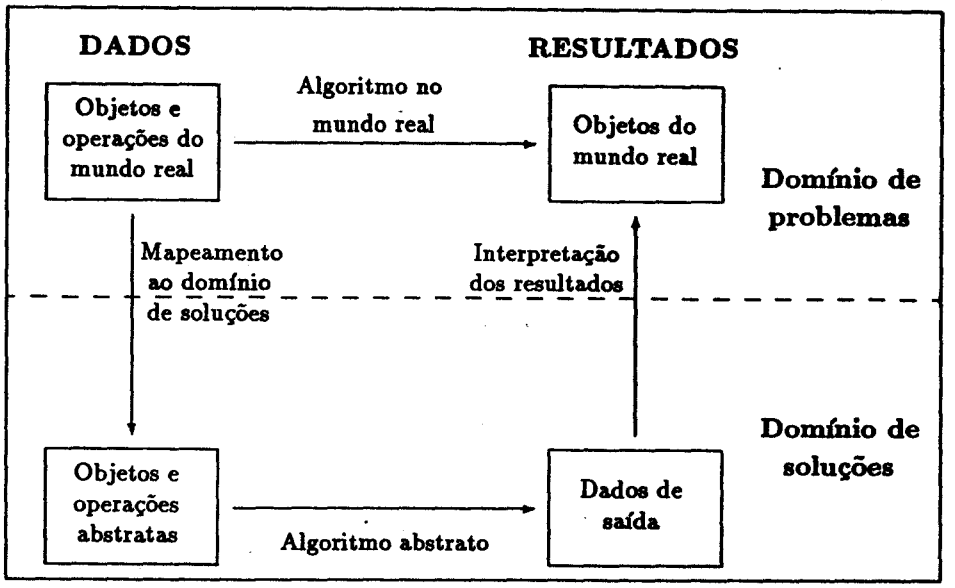

Figura 2.1: O *"gap"* semântico (adaptado de [Tak89])

2.2.2 Operações abstratas

O uso correto do mecanismo de abstração de dados, descrito na seção anterior, constitui a essência da atividade de desenvolvimento de software no paradigma de objetos.

No desenvolvimento de uma aplicação, o projetista de software observa a realidade do problema e dela **abstrai** as entidades e os fenômenos que ele julgar relevantes para formar o chamado **modelo** da aplicação. Por exemplo, suponhamos que está sendo analisado o mapa de uma cidade. Esse mapa representa os aspectos de uma realidade (a cidade) e dele, dependendo do **objetivo** de quem o vê, pode-se abstrair diversos domínios. Esse mapa pode identificar, para um determinado indivíduo, o caminho que deve ser percorrido entre dois pontos (entre duas ruas), ignorando-se outros detalhes presentes no mapa, tais como o nome dos bairros, o nome das ruas adjacentes, etc.; enquanto que, para um outro indivíduo, a informação que se deseja abstrair é o nome dos bairros próximos ao seu, ignorando-se outros detalhes como por exemplo, o nome das ruas, os bairros mais distantes, etc. Ou seja, somente as informações relevantes para resolução de um determinado problema é que são consideradas para efeito de identificação do domínio da aplicação através de operações mentais.

Visto como a abstração de um domínio é a primeira grande atividade da tarefa de desenvolvimento de software, o próximo passo seria o de **identificar** quais as operações passíveis de serem realizadas sobre entidades de um domínio de problemas. Pesquisas na

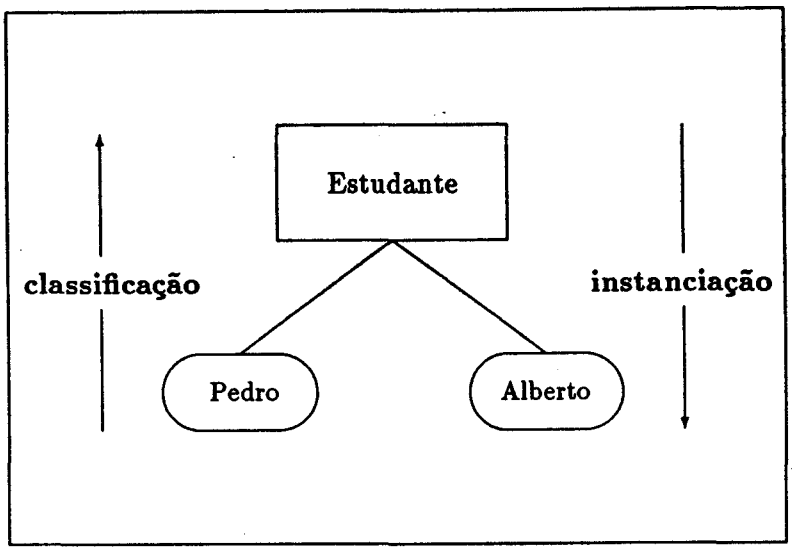

Figura 2.2: As operações abstratas de classificação e instanciação

área de banco de dados identificaram basicamente **6 operações abstratas** básicas sobre domínios:

- classificação
- instanciação
- generalização
- especialização
- agregação
- refinamento

Para ilustrar o uso destas operações, serão adotados como exemplo os agentes formadores de uma universidade (professores, funcionários e alunos) e, de acordo com o nível de abstração a que se fará menção, aqueles agentes poderão ser caracterizados com maiores detalhes[4] (ex. os alunos poderão ser redefinidos para incluir os alunos de graduação e os alunos de pós-graduação).

Através da operação de **classificação** entidades similares são associadas a grupos (classes), em função de propriedades comuns existentes entre estas entidades. Na figura

[4] especializados.

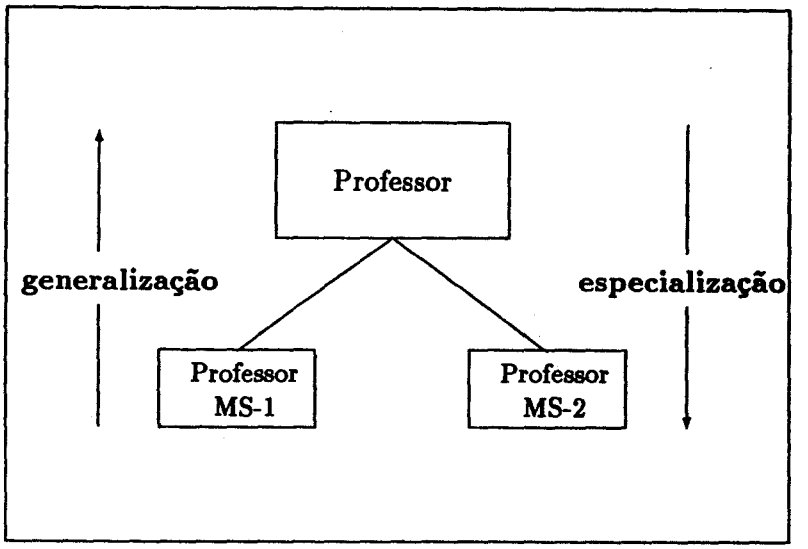

Figura 2.3: As operações abstratas de generalização e especialização

2.2 é ilustrada a operação de classsificação, onde temos dois indivíduos (Pedro e Alberto) sendo **associados** à classe estudante. A classe estudante representa de forma abstrata as propriedades comuns aos dois estudantes, como, por exemplo, terem um RA (registro acadêmico), poder freqüentar o restaurante universitário, etc.

A operação inversa à de classificação é a **instanciação**. Através dela, instâncias ou **objetos** de uma classe podem ser criados e definidos. Estas instâncias possuem as características descritas de forma abstrata na classe da qual foi instanciada.

A próxima operação a ser identificada é a de **generalização**. Esta operação é importante para identificar e formar uma **hierarquia de abstrações** das entidades a serem representadas e pode ser aplicada quando duas classes possuem algumas propriedades semelhantes. Estas podem ser abstraídas ("fatoradas") numa classe mais genérica. A classe mais genérica é considerada uma **generalização** das duas primeiras. Na figura 2.3 temos um exemplo da operação abstrata de generalização. Temos, a generalização das classes Professor MS-1 e Professor MS-2 na classe Professor. Uma das características que poderiam ser **generalizadas** na classe Professor seria, por exemplo, ter de comparecer à reunião do departamento, enquanto que uma das propriedades que permaneceria nas classes especializadas seria a de ter o curso de graduação completo (para a classe Professor MS-1) e um curso de pós-graduação completo (para a classe Professor MS-2).

A operação inversa à de generalização é a de **especialização**. Por esta operação, classes **mais específicas** de classes **mais abstratas** são criadas e definidas. Neste caso as classes mais específicas herdam propriedades das classes mais gerais das quais foram

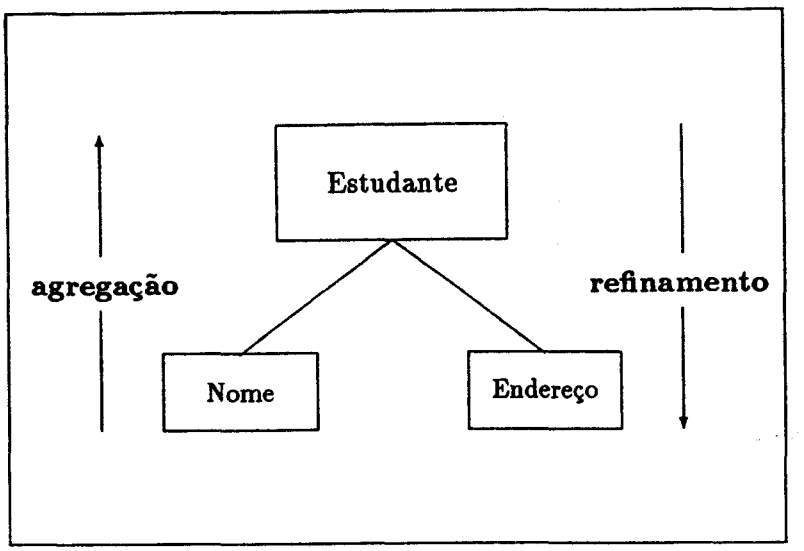

Figura 2.4: As operações abstratas de agregação e refinamento

derivadas.

As últimas operações abstratas na modelagem de um domínio são a **agregação** e o **refinamento**. A agregação é uma operação que permite que uma nova categoria possa ser criada, a partir de composições ou agregações de outras categorias. Na figura 2.4 as operações de agregação e refinamento são ilustradas através das classes Estudante, Nome e Endereço[5]. A categoria Estudante foi formada através da **agregação** de instâncias das classes Nome e Endereço. Esta operação faz com que as entidades possam ser agregadas sucessivamente em entidades maiores, e detalhes se vão tornando irrelevantes a cada novo passo. A este processo dá-se o nome de agregação ou **composição sucessiva**. O processo inverso, em que as entidades vão sendo a cada passo mais detalhadas dá-se o nome de refinamento ou **decomposição sucessiva**.

2.2.3 Hierarquia de abstrações

Uma hierarquia de abstrações nada mais é do que a representação abstrata de entidades do domínio construído através das operações abstratas ilustradas na seção anterior.

O conceito de hierarquia de abstrações será ilustrado através de um quadro de pessoal de uma empresa hipotética da área de informática. A figura 2.5 mostra o quadro em que se deseja formar a hierarquia de abstrações.

[5]será levado em consideração, para este exemplo, que os atributos nome e endereço são na realidade duas classes representando o nome e o endereço de indivíduos.

diretores: presidente, vice-presidente, financeiro

chefes: de departamento, de pessoal

líderes: de projetos

analistas

programadores: júnior e senior

documentadores

funcionários: contador, servente, digitador, secretária, auxiliar de escritório.

Figura 2.5: Quadro de pessoal de uma empresa da área de informática

Uma primeira análise sobre este **domínio** leva uma grande parte de projetistas de aplicações, habituados a lidar com desenvolvimento procedural, a construir a **hierarquia** de abstrações, como mostrado na figura 2.6. Esta análise leva a um absurdo de raciocínio se for considerado o mecanismo de herança. Analisando a figura 2.6 é possível concluir que, para esta "hierarquia", a categoria digitador **herda** todas as características da categoria analista e, mais aberrante ainda, um funcionário qualquer possui, dentre as suas características, as atribuições de um diretor presidente. Na verdade, esta **indução** ao erro é muito comum e mostra a completa inversão de pensamento a ser seguida na análise do domínio de uma aplicação desenvolvida sob o paradigma de objetos. A figura 2.6 poderia ser a representação de um **organograma** de uma empresa e não uma hierarquia de abstrações.

A figura 2.7 ilustra, então, uma possível hierarquia para descrever os indivíduos que operam nesta empresa. As classes estão representadas por retângulos e a interligação entre elas é feita através de um segmento de reta contínuo representando as operações abstratas de **especialização**[6] e **generalização**[7]. Isto significa dizer que a classe digitador é uma **especialização** da classe funcionário. Ou ainda a classe líder de projeto é uma **generalização** da classe chefe de departamento. As intâncias derivadas de classes desta hierarquia (pessoas formadoras do quadro de pessoal - Eduardo, Beatriz, etc.) estão representados por retângulos ovalados, e a operação abstrata de **instanciação** é representada

[6] na análise *top-down*.

[7] na análise *botton-up*.

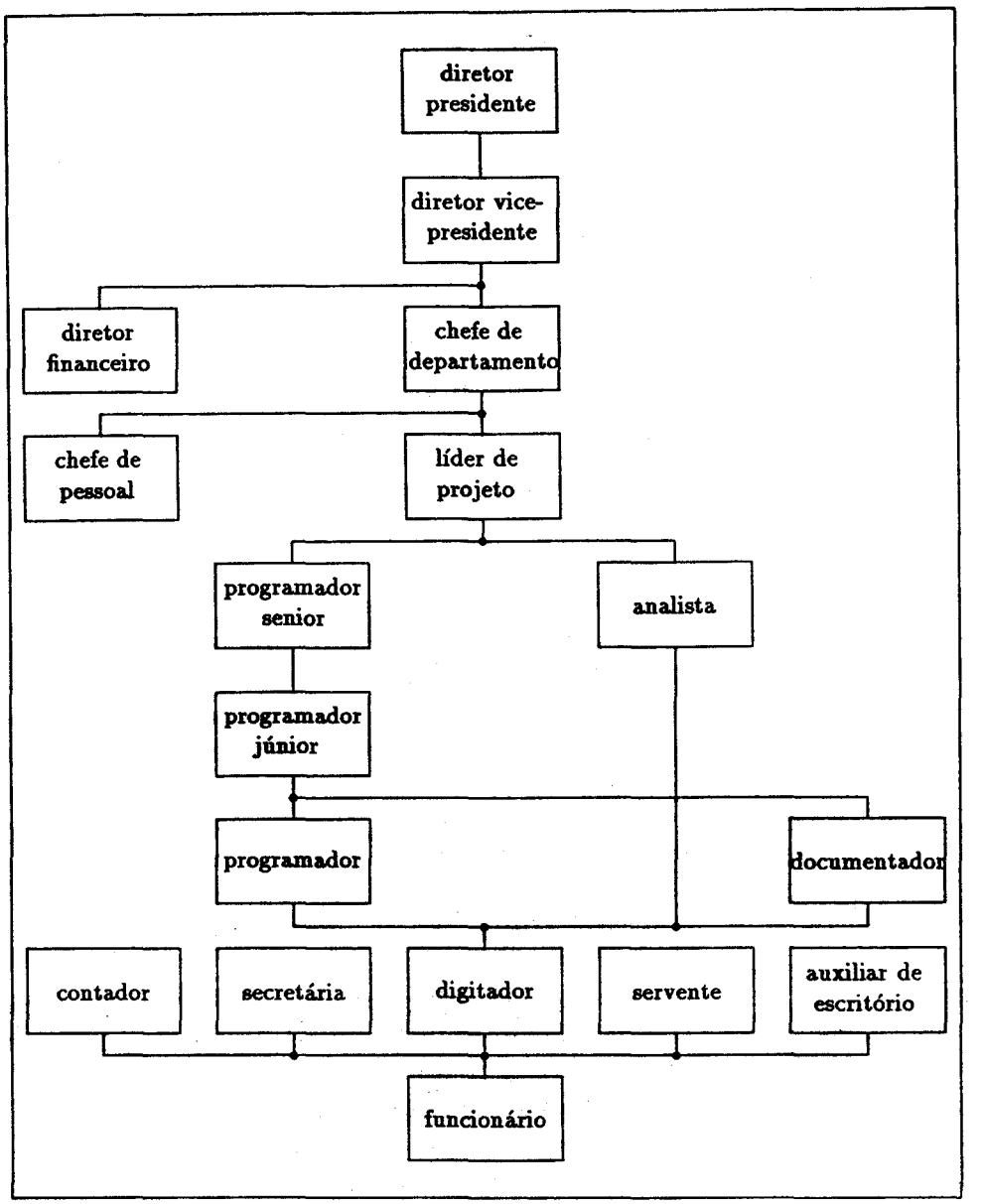

Figura 2.6: Um organograma de uma empresa da área de informática

por um segmento de reta contínuo entre a classe da qual foi derivada e a instância (nesta ordem). A operação inversa da instanciação, que é a **classificação**, pode ser observada se a análise for realizada no sentido inverso: da instância para a classe. Desta forma, isto significa dizer que o objeto Eduardo é o diretor presidente da companhia pois ele foi **instanciado** a partir da classe diretor presidente; e que o objeto Bruno é um dos programadores seniors, pelo mesmo motivo.

2.3 Suporte para programação orientada a objetos

2.3.1 Características das linguagens

O suporte à abstração de dados descrito na seção anterior, mesclado ao mecanismo de herança, e o uso conveniente do mecanismo de acoplamento (precoce e tardio) entre mensagem e método fazem com que uma linguagem de programação proveja o suporte básico à orientação objetos.

As linguagens desenvolvidas para suportar este paradigma devem incorporar as modificações **fundamentais** sugeridas pelo novo estilo. Em primeiro lugar, as linguagens orientadas a objetos acrescentam estruturas sintáticas que permitem a representação de tipos abstratos de dados (a própria definição do que vem a ser uma **classe**). Além disto, elas incorporam um estilo de desenvolvimento que permite uma forte interação gráfica com os usuários e, a nível metodológico, elas absorvem uma nova abstração para a arquitetura de sistemas, através da implementação dos conceitos de hierarquias de tipos abstratos de dados (herança de classes) e, conseqüentemente, a modularização do sistema sendo desenvolvido.

Em relação à usabilidade da nova linguagem, seria **utopia** afirmar e garantir a migração de usuários de outras plataformas para o estilo orientado a objetos, sem que houvesse um forte apelo em relação à **performance** do novo sistema. As linguagens orientadas a objetos (ou qualquer outro tipo) devem, em primeiro lugar, "tentar" competir em performance com as linguagens mais convencionais e, se possível, oferecer além dos atrativos metodológicos e lingüísticos, um suporte para uma migração mais flexível e menos problemática (modificações complexas de módulos).

Uma resposta para este problema vem sendo adotada para os adeptos do paradigma procedural e funcional. As linguagens C++, Object-Pascal e CLOS, por exemplo, são uma espécie de *super-set* das linguagens procedurais C, Pascal e da linguagem funcional Common LISP, respectivamente. Em todas elas, a versão do paradigma orientado a objetos incorpora as mudanças fundamentais do novo estilo.

Dentro do paradigma de objetos, diversas linhas têm surgido e vêm sendo solidificadas em relação ao **suporte a objetos** que elas provêem.

- a linha de **objetos passivos** - é o suporte mais difundido e testado. Baseia-se na adoção do princípio de que um objeto se torna ativo quando recebe uma **mensagem**

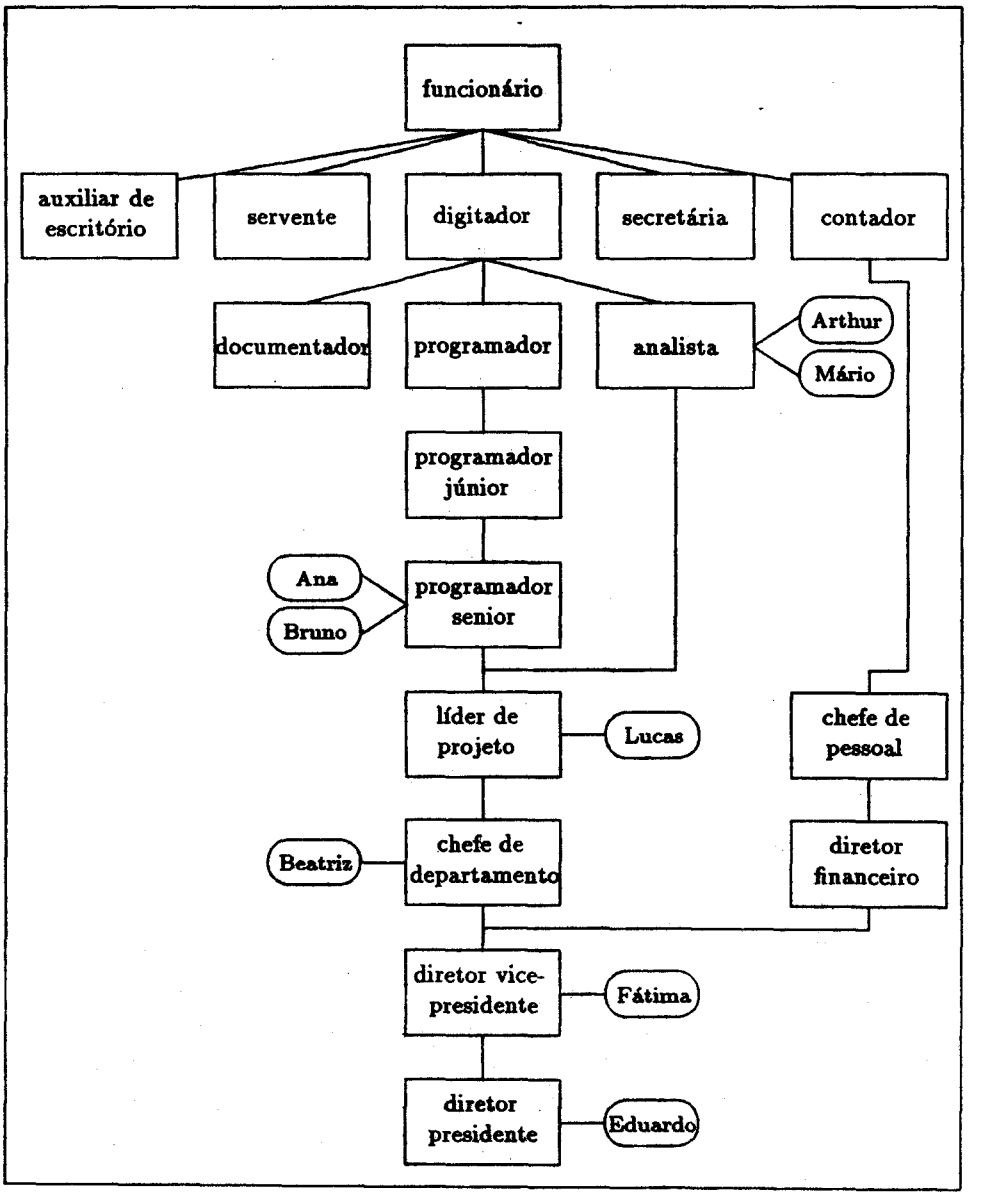

Figura 2.7: Uma hierarquia de abstrações de uma empresa da área de informática

síncrona e passa a ser passivo novamente após enviar a resposta decorrente do processamento da mensagem ao remetente da mesma. O remetente é passivo enquanto o destinatário completa a resposta e ao receber a resposta a uma mensagem enviada, passa a ser ativo novamente. Durante qualquer outra fase ele permanece passivo e pronto para ser "acordado". Esta linha originou a maior parte das linguagens até agora implementadas. é o caso de Smalltalk, C++, Object-Pascal, dentre outras. Esta é a linha que será adotada para realização e implementação do projeto descrito neste trabalho;

- a linha de **objetos ativos e objetos concorrentes** - este suporte baseia-se no princípio de que os objetos têm comportamento dinâmico e autônomo. Nesta linha, um objeto ao receber uma **mensagem** torna-se ativo e, dependendo do tipo de **tarefa** a ele incubido, uma **resposta** poderá ser enviada ao objeto remetente, mesmo sem que a tarefa tenha sido executada completamente. Esta **concorrência** de atividades tem sido discutida em artigos como em Fiume [Fiu89] e algumas linguagens já possuem protótipos em disponibilidade. é o caso das linguagens ABCL/1 e POOL-T (Parallel Object Oriented Language) ambas também citadas em Fiume [Fiu89];

- a linha de **objetos evolutivos** - provavelmente o suporte mais contemporâneo de todas as linhas do paradigma de objetos. é a linha que mais se aproxima das **definições reais** do que vem a ser um objeto (uma espécie de "ente-vivo" com memória privada e um comportamento volátil). Nesta linha, um objeto pode ser dinamicamente (re)definido, não só através da adição de novos **métodos** mas também através do acréscimo de propriedades (**atributos**) à sua **memória interna**. Além disso, os objetos definidos nesta linha, podem tanto agregar quanto perder atributos e comportamentos de outras classes já definidas. Esta **evolução dos objetos** tem uma analogia muito comum e é freqüêntemente citada para ilustrar este suporte: o ser humano homem (o objeto evolutivo) não nasceu adulto! Durante alguma etapa de sua existência ele **adquiriu** as propriedades de adulto (adição de atributos à sua memória interna) e perdeu as características de adolescente. Em uma outra ocasião ele tornou-se pai. Isto significa dizer que ele incorporou ao comportamento de adulto as **propriedades herdadas** do comportamento de um pai (adição de novos métodos). Esta é a linha que mais se aproxima das definições reais do que deveria ser um objeto. Pois como todo "ente-vivo", modificações e evoluções deveriam ser incorporadas ao seu estado interno de uma forma "natural", durante toda a existência de um objeto. Esta evolução de propriedades em linguagens de programação tem sido propostas, por exemplo, em Flavors e CLOS.

Em qualquer das linhas adotadas, é consenso afirmar que uma linguagem de programação, para poder ser dita orientada a objetos, deve permitir a definição de tipos abstratos de dados na forma de classe e permitir definir hierarquias de classes através de

mecanismos de herança. Além disto, outras propriedades são **desejáveis**, tais como as que serão descritas na seção seguinte.

2.4 Conceitos Básicos

2.4.1 Introdução

Os conceitos básicos do paradigma de objetos expostos nesta seção, como dito no início do capítulo, serão introduzidos segundo a linha de **objetos passivos**. A terminologia do paradigma é composta de alguns conceitos, já citados propositalmente (outros não) em seções anteriores, e agora elucidados.

Em algumas situações serão ilustrados exemplos de definições destes termos utilizando-se para tal trechos de código na linguagem C++ visto que nesta última foi feita a implementação do projeto descrito no capítulo 4.

Desenvolver uma aplicação sob o paradigma de objetos significa dizer que esta aplicação será uma representação de um modelo do mundo real, composta por **agentes**, que se relacionam entre si. Estes agentes são representados por **objetos** que podem ser comparados (grosseiramente) com as variáveis de um programa procedural convencional; os objetos são instanciados a partir de **classes** "comparáveis" aos tipos do paradigma procedural e se comunicam e trocam **mensagens** entre si que poderiam ser associadas a chamadas de procedimentos.

2.4.2 Classes

A primeira (e maior) tarefa na representação de um modelo em computador é a definição **de quais** classes são necessárias para um determinado domínio. A definição de uma classe pode ser comparada à própria definição de um **tipo abstrato de dado**. Através dela é possível traduzir em "um tipo" o estado interno e o comportamento de objetos com caractetísticas semelhantes.

Uma classe determina:

- o nome do "tipo" de objeto que ela representa;
- os atributos (**dados**) associados a este objeto;
- os métodos que são internos ao objeto;
- o comportamento inicial (e final) que um objeto deve ter na sua criação (e morte);
- a visibilidade externa do **comportamento** deste objeto;
- as classes das quais é **derivada** (se houverem) diretamente.

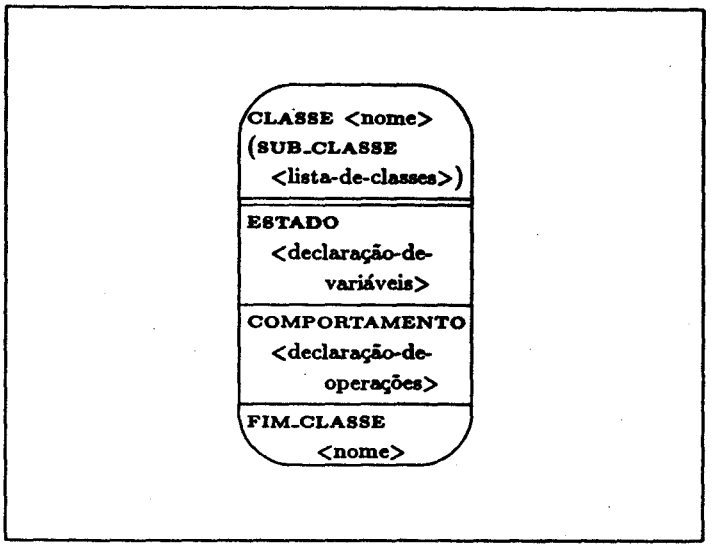

Figura 2.8: Molde de uma classe (adaptado de [TLX90])

Uma convenção simples, como a adotada em Takahashi [TLX90] é mostrada na figura 2.8, e serve para descrever a representação do molde de uma classe.

Nesta convenção, as letras em maiúsculas e negrito são palavras reservadas, e as palavras em minúsculas e entre "<" e ">" são identificadores a serem substituídos na utilização do molde. Os símbolos entre parênteses são opcionais.

Um exemplo de utilização do molde de uma classe poderia ser como ilustrado na figura 2.9. Serão utilizados exemplos derivados de [Tak89], cujas classes são formadas por figuras do mundo animal, para ilustrar a utilização do molde de uma classe. As duas classes mamífero e cão têm algumas definições e comportamentos similares (por exemplo: ambos são vertebrados, têm dentes e na fase pós-gestação se alimentam através de leite materno) e, na seção 2.4.5, quando a propriedade de **herança** de classes for ilustrada, esta similaridade de propriedades será importante para modelar a aplicação.

As classes mamífero e cão definem as características de objetos que serão instanciados e que possuirão comportamentos idênticos[8]. A classe mamífero é também chamada de **classe base**[9], pois está localizada na raiz de uma hierarquia (não é especialização de

[8] a princípio todos os mamíferos se defenderão quando molestados, ou então todos os cães latem de maneira idêntica. Quando houver necessidade de se diferenciar objetos dentro de uma mesma classe (ex. cão doberman e cão puddle) novas classes devem ser criadas (como especializações da primeira) para refletir esta nova alteração.

[9] do inglês *base class*.

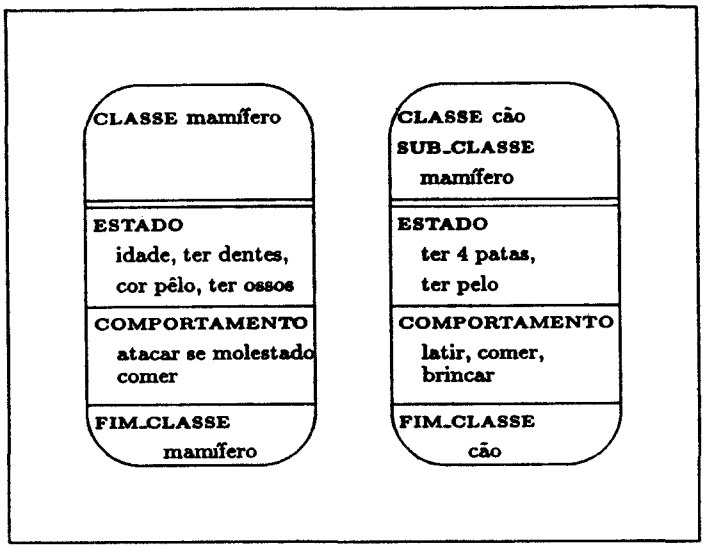

Figura 2.9: Exemplo de classes

nenhuma classe) e a classe cão é chamada de **classe derivada**[10] pois é uma **sub-classe** de uma classe já definida na hierarquia.

Os mamíferos que têm comportamento idênticos têm seus **métodos** definidos na classe base. E, quando houver necessidade de diferenciação entre dois tipos de animais (ex. o **cão** se comunica de uma forma diferente da do **gato**), novas classes especializadas devem ser derivadas através da agregação de atributos e comportamento mais específicos. A especialização é um processo incremental. Características gerais são herdadas de superclasses enquanto que características específicas são adicionadas.

Um exemplo mais concreto (e mais técnico) sobre o uso de classes é mostrado na figura 2.10.

A classe shape é a classe base de todas as figuras geométricas (círculo, quadrado, retângulo, etc.) e nela são definidas as variáveis (o estado interno) e os métodos (comportamento) que são comuns a todas as figuras (*i.e.* toda figura geométrica tem um **centro** e uma **cor**).

Este exemplo (e modificações sobre ele) será mencionado nas subseções seguintes deste capítulo, uma vez que o mesmo será usado para ilustrar novos conceitos.

A figura 2.11 ilustra a representação da classe shape na linguagem C++. A introdução de novas construções sintáticas será feita de maneira **intuitiva**; uma completa definição destas pode ser encontrada em [Str86, Ber89, Mic89d].

[10] do inglês *derived class*.

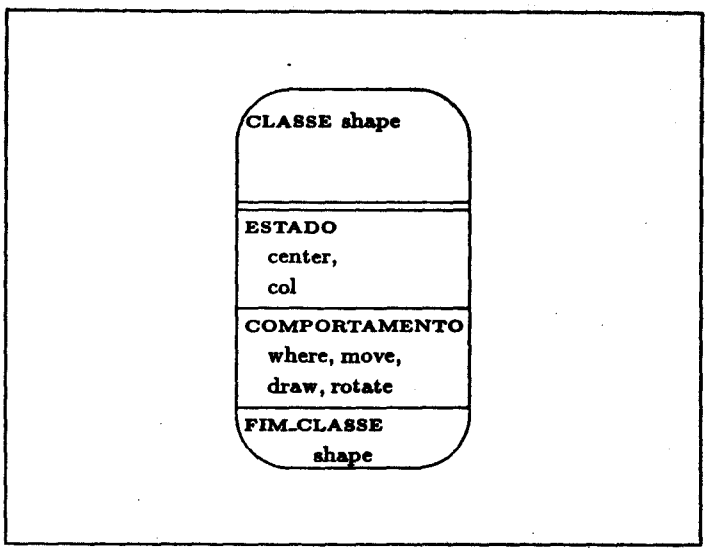

Figura 2.10: A classe shape

```
class shape {
   point center;
   color col;
public:
   void where(void);
   void move(point);
   virtual void draw(void);
   virtual void rotate(int);
};
```

Figura 2.11: Declaração da classe shape em C++ (adaptado de [Str88])

2.4.3 Objetos

Um objeto é uma instância de apenas uma classe. No início do capítulo foi feita uma analogia a um objeto como sendo este um "ente vivo" que possui estado interno e comportamento próprios. Para obter esta "vida", um objeto necessita ser criado ou instanciado. Em C++, a sua instanciação pode ser realizada de duas maneiras diferentes:

- através de uma instanciação
  ex. `shape s1;` ou
- através do operador `new`
  ex. `shape *s1 = new shape();`.

A partir de sua criação, o objeto permanecerá vivo até a execução do programa passe para fora do escopo da variável que identifica o objeto ou, no segundo caso, até que seja explicitamente eliminado (através do operador `delete`).

2.4.4 Mensagens e métodos

A comunicação entre objetos que formam um programa é realizada através de **mensagens**. A analogia que foi citada anteriormente, de que um objeto é um "ente vivo", leva à constatação de que todo "ente vivo" precisa ser **visível** e **estimulável**. A troca de **mensagens** entre os objetos de uma aplicação ocorre pela necessidade de se tornar os objetos **visíveis** e **estimuláveis**. Na subseção 2.4.10 e no capítulo 4, que ilustram e definem o uso do **princípio reativo** na construção de interfaces, este conceito de **objetos visíveis** e **estimuláveis** será redefinido a fim de agregar o suporte a **múltiplas visões**.

A princípio, um objeto torna-se **visível** quando estimulado a atuar sobre o seu estado interno através de uma **mensagem**. O objeto que recebe a mensagem responderá à mesma através da seleção e da execução de um **método** que implementa a reação do objeto a esta mensagem específica.

É possível implementar em linguagens orientadas a objetos a propriedade de que objetos instanciados a partir de classes distintas reajam de forma especializada a uma mesma mensagem. A esta propriedade dá-se o nome de **polimorfismo** e o seu uso será descrito na subseção 2.4.7.

Uma outra característica relacionada com o envio/recebimento de mensagens é que o objeto que envia a mensagem, neste modelo passivo, permanece parado ou **inativo**, até que aquela mensagem seja **respondida** pelo outro objeto.

Na figura 2.12 é ilustrado envio de mensagens entre objetos instanciados a partir da classe shape. é interessante notar que uma mensagem é composta de um seletor e de eventuais parâmetros (dependendo da definição e da interface com o método sendo invocado).

A figura 2.13 ilustra a implementação de algumas **mensagens** definidas na classe shape.

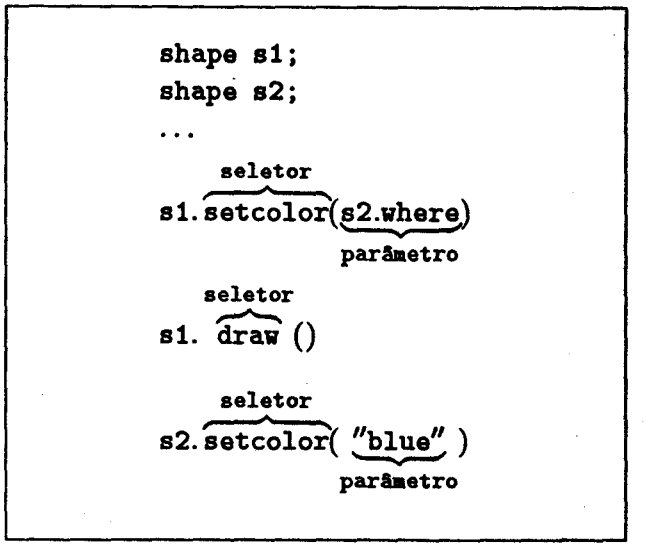

Figura 2.12: Exemplos de envio de mensagens

```
point shape ::  where(void)
{
   return center;
}

void shape ::  move(point to)
{
   center = to;
   draw();
}
```

Figura 2.13: Exemplos de alguns métodos da classe shape

2.4.5 Herança

A propriedade de **herança** de classes é, sem dúvida, a característica mais importante das linguagens do paradigma de objetos. A idéia básica deste conceito é a de que as classes representadas numa hierarquia de abstrações delegam atributos e comportamento menos específicos para subclasses mais especializadas.

Na hierarquia de abstrações da figura 2.7, por exemplo, as classes digitador e secretária herdam todas as propriedades da classe funcionário, e isto equivale a dizer que as classes digitador e secretária são **subclasses** da classe funcionário e esta é dita **superclasse** das outras duas. Na operação de modelagem de domínios, o fator de herança deve ser considerado e, na implementação destas classes, as propriedades comuns a digitadores e secretárias[11] (por exemplo, todos os digitadores e secretárias têm que assinar um livro de ponto ou ambos possuem folga semanal, etc.) podem ser abstraídas em uma superclasse que então repassa estas características às suas subclasses via mecanismo de herança. Em relação ao programador da aplicação, este repasse se dá de maneira transparente e os métodos definidos e herdados da superclasse podem ser usados na subclasse, da mesma forma que os métodos específicos definidos na própria subclasse[12].

Na figura 2.14 é ilustrado o mecanismo de herança em relação as classes shape (definida na figura 2.10), circle e square.

A classe circle é uma subclasse da classe shape e a ela são adicionados: um novo atributo (radius) e um novo método (area); será redefinido um método (draw) e herdados outros (move, rotate, where). Na figura 2.14 o método draw foi **redefinido**. Neste caso particular quando do envio da **mensagem** draw para um objeto instanciado da classe circle, o método draw mais específico (o da classe circle) será executado. Em relação ao método que foi adicionado (area), a execução funciona da mesma forma que os métodos de uma classe normal como shape: o envio da mensagem area a um objeto da classe circle irá fazer com que o método específico e único definido nesta classe seja executado. Por fim, o envio de uma mensagem definida em shape (move, where, rotate) e enviada a um objeto instanciado de circle faz com que as versões dos métodos destas mensagens **definidos na superclasse** sejam executados.

Na figura 2.15 é mostrada a especificação e implementação da classe circle. Em C++ o mecanismo de herança pode ser utilizado, como mencionado, de duas maneiras: através da herança pública e da herança privada.

Na herança pública (a da figura 2.15) os métodos derivados são incorporados à nova classe em definição, com o mesmo grau de visibilidade com que foram definidos na superclasse. Então, se um método foi definido com `protected`[13] na superclasse, será in-

[11] ou seja as propriedades da classe funcionários.

[12] C++ "modifica" este conceito através da introdução de dois tipos de herança de classes - a herança pública e a herança privada - que pode propagar ou restringir a herança de métodos.

[13] na seção seguinte deste capítulo, quando forem ilustrados os conceitos de **ocultamento de dados**, os três níveis de proteção aos dados de uma classe (`private`, `protected` e `public`) serão definidos.

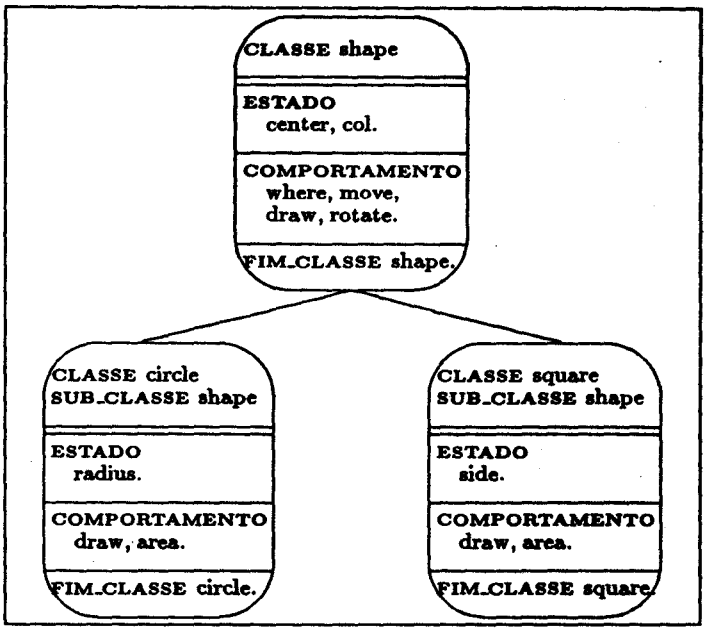

Figura 2.14: Uma hierarquia de figuras geométricas

```
class circle :  public shape {
   int radius;
public:
   void draw(void);
   float aread(void);
};
...
void circle ::  draw(void) {
   // code for draw
}
float circle ::  area(void) {
   return 3.14 * radius * radius;
}
```

Figura 2.15: Especificação e implementação da classe circle em C++

corporado na subclasse como `protected` (protegido). Da mesma forma que os métodos definidos como `public` (públicos) ao serem herdados, serão incorporados com o mesmo grau de proteção e visibilidade da superclasse.

Na herança privada[14], **todos** os métodos definidos na superclasse (com grau de proteção `protected` e `public`) são introduzidos na subclasse como `private` (privado) e, desta forma, possuem as mesmas propriedades de visibilidade que atributos e métodos definidos nesta seção.

O uso do mecanismo de herança na definição de classes leva a um paradoxo: uma classe definida na **base**[15] de uma hierarquia de abstrações herda as características de **todas** as classes (da mais específica à mais genérica) subordinadas a esta. Isto leva à conclusão de que, apesar do uso da herança levar, dentre outras vantagens, a uma **reutilização** grande de software, o número de métodos para uma classe **bem específica** pode ser demasiadamente grande, e provavelmente boa parte destes métodos não serão mais relevantes (ex. na hierarquia da figura 2.7, a classe diretor presidente herda as características da classe funcionário e propriedades deste tais como assinar o cartão de ponto na chegada e o recebimento de vale refeição, propriedades que provavelmente **nunca** serão usadas). Constatado este fato, seria interessante que na propagação de características via herança, além da adição e substituição de propriedades, existisse um mecanismo que implementasse o **cancelamento** de métodos. O modelo definido para o C++ **não** incorpora a característica de **cancelamento de propriedades** na herança de classes.

2.4.6 Ocultamento de dados

No paradigma de objetos classes agregam atributos que definem o estado de objetos derivados destas classes bem como operações sobre este estado denominadas métodos. Os métodos, portanto, definem o protocolo de uso de objetos. Geralmente só é permitido a manipulação de atributos via métodos.

Um aspecto relacionado com o encapsulamento provido por classes é o que descreve o controle da visibilidade de atributos em relação a subclasses especializadas da classe em questão e em relação aos métodos definidos na própria classe.

Em C++ este controle de visibilidade é especificado através das seções de uma classe denominadas `private`, `protected` e `public`.

O grau de visibilidade em cada uma destas seções é o seguinte:

- Seção privada (`private`) - os métodos e atributos definidos nesta seção são visíveis **apenas** para as funções membros (nomenclatura de C++ para denotar métodos) definidas nesta classe. Fica vedado, portanto, o acesso por métodos de outras classes (derivadas ou não) e para objetos de uma forma geral;

[14] utilizada da mesma maneira que na pública só que sem a palavra reservada `public`.

[15] a palavra base, neste sentido, significa que a classe foi definida em um nível de especificação o mais especializado possível de uma hierarquia.

- Seção protegida (`protected`) - a visibilidade de atributos e métodos definidos nesta seção é ampliada para os métodos de todas as subclasses diretas e indiretas da classe em questão;

- Seção pública (`public`) - a visibilidade das informações (geralmente apenas métodos) definidas nesta seção é ampliada para qualquer parte do código de um sistema que manipula objetos derivados direta ou indiretamente da classe em questão.

A figura 2.16 apresenta de maneira esquemática uma hierarquia de abstrações com a definição de três classes (parent, child1 e child2).

A classe parent é a **classe base** desta hierarquia e possui um objeto (p) instanciado. A classe child1 é uma subclasse especializada através de **herança privada** da classe parent. A classe child2 também é uma subclasse especializada da classe parent através da **herança pública**. Os dois objetos (c1 e c2) foram instanciados de child1 e child2, respectivamente.

2.4.7 Polimorfismo

O polimorfismo é uma propriedade de linguagens orientadas a objetos que confere um certo grau de versatilidade ao código produzido, uma vez que a sua utilização permite a escrita de código genérico. O conceito de polimorfismo pode ser resumido como sendo:

> A reação especializada de objetos a classes distintas a mensagens idênticas.

A utilidade desta facilidade pode ser ilustrada através do exemplo de figuras geométricas mencionado no início desta seção.

As figuras geométricas (círculo, quadrado, pentágono,etc.) criadas através de um editor de figuras (para ilustração deste exemplo) estão representadas, na arquitetura interna deste editor, em uma "lista ligada" de objetos do tipo shape. Em um determinado momento, a função **redraw** do editor foi acionada, significando que todas as figuras, até o momento projetadas, precisam ser redesenhadas. Um código em **linguagem algoritmica** que este editor poderia acionar para redesenhar cada figura desta lista poderia ser como o ilustrado na figura 2.17.

Uma observação importante neste algoritmo é a ausência de identificação da natureza da figura que esta lista armazena. Ou seja, qualquer que a figura encontrada na lista, o método draw será elicitado, e o seu código, definido na implementação da classe da qual a figura instanciada, será executado. A vantagem que esta propriedade introduz é a possibilidade de confecção de métodos e procedimentos genéricos como o da figura 2.17. Se, em uma outra ocasião, for criada uma figura geométrica instanciada de uma nova classe e incorporada à lista, este procedimento **permanecerá imutável**. A simplicidade

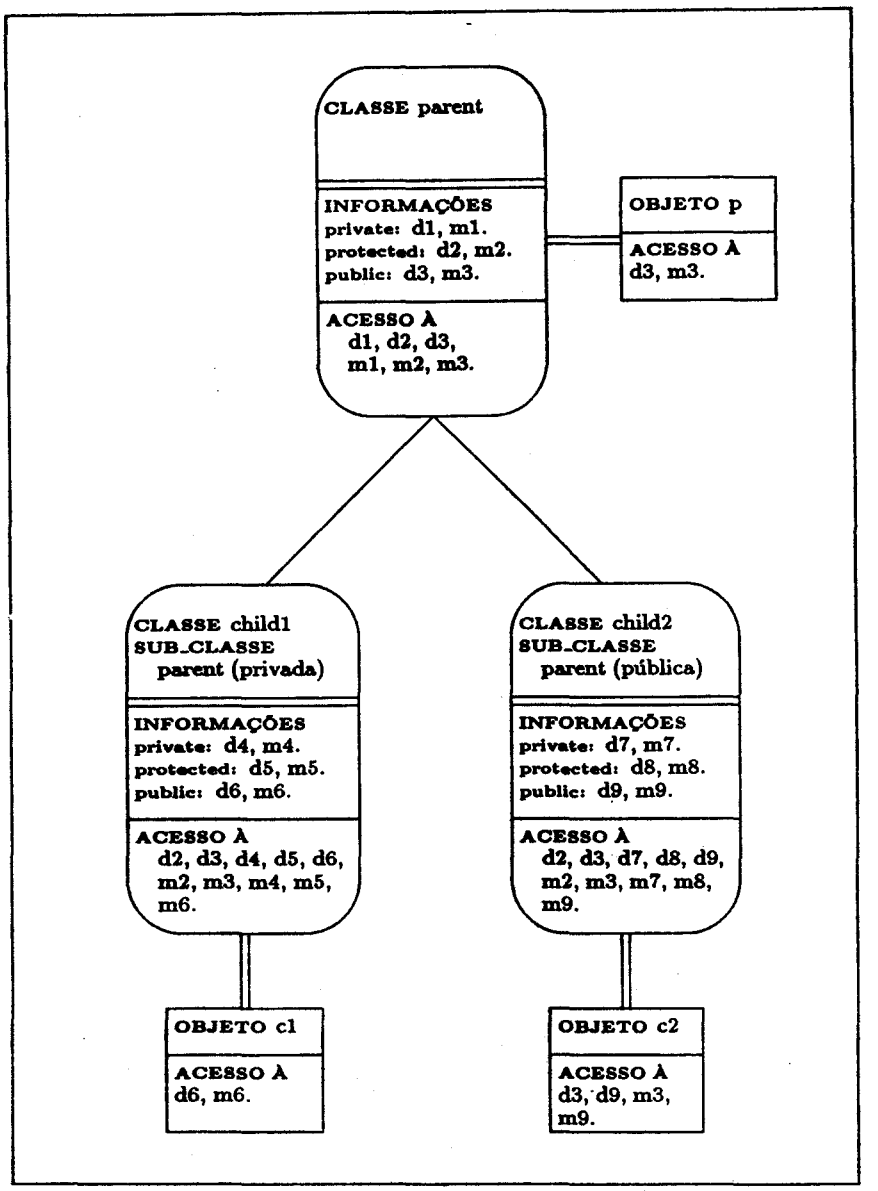

Figura 2.16: Uma hierarquia de abstrações e acessos permitidos

```
while (!empty(shape_list))
{
   (shape_list.figure).draw();
   next(shape_list);
}
```

Figura 2.17: Algoritmo para redesenhar as figuras geométricas de uma lista

e clareza deste procedimento baseiam-se no **acoplamento dinâmico** entre mensagem e método, tal como será descrito na subseção seguinte.

Em linguagens cujo acoplamento entre as mensagens enviadas e os métodos executados seja realizado de maneira estática[16], este procedimento teria que ser modificado[17], tal como o mostrado na figura 2.18. é interessante notar que a inserção de qualquer nova figura geométrica à lista acarreta uma modificação no algoritmo (uma nova entrada no `switch`).

2.4.8 Acoplamento (precoce e tardio) mensagem/método

A comunicação entre os objetos de um programa se dá, como dito, através do envio de **mensagens** e da execução de **métodos** correspondentes. Um fator importante de linguagens baseadas neste paradigma é o modo de como é efetuada a ligação[18], entre a mensagem sendo enviada e o método a ser executado. Em tese, é interessante que as linguagens promovam suporte para que o acoplamento entre mensagem e método possa acontecer por opção do programador[19], tanto em tempo de compilação (acoplamento **precoce**, quanto em tempo de execução (acoplamento **tardio**).

No caso do acoplamento precoce[20], o compilador consegue identificar, durante a geração de código, qual o método a ser executado em função do envio de uma mensagem a um objeto explicitado no código submetido ao compilador. O "pior" caso para este tipo de ligação é o de existirem na classe do objeto em questão diversos métodos com o mesmo nome para o seletor da mensagem. O compilador terá que realizar uma análise sobre os argumentos, para "decidir" qual será o método a ser invocado. A esta possibilidade de

[16] provavelmente uma linguagem que só possua o acoplamento estático nem seja caracterizada como orientada a objetos. O acoplamento dinâmico (ou pelo menos híbrido) é um requisito intrínseco do paradigma de objetos.

[17] ou melhor: descaracterizado.

[18] acoplamento, *binding*.

[19] acoplamento seletivo.

[20] da expressão inglesa *early binding*.

```
while (!empty(shape_list))
   switch (shape_list.kind)
   {
      case triangle:  //call method draw in triangle
                      break;
      case square:    //call method draw in square
                      break;
      ...
      case circle:    //call method draw in circle
                      break;
   }
   next(shape_list);
}
```

Figura 2.18: Algoritmo para redesenhar as figuras geométricas de uma lista

atribuir a métodos distintos o mesmo identificador distinguíveis apenas pelos tipos e/ou número distintos de seus parâmetros dá-se o nome de sobrecarga[21] de funções e seu uso será ilustrado, em C++, na seção 2.4.9.

A figura 2.19 ilustra o envio e o acoplamento de mensagens em C++, na forma precoce.

O acoplamento tardio é o tipo de ligação que permite escrever código mais flexível visto que, em tempo de execução, sempre o código mais especializado é executado. Este fato é relevante para envio de mensagens para objetos referenciados indiretamente. No caso particular do C++, um apontador associado a uma classe pode referenciar objetos instanciados desta classe ou de suas subclasses. Neste caso não é possível identificar, em tempo de compilação, qual o objeto referenciado pelo apontador. Quando é adotada a ligação estática, o método mais específico não será necessariamente executado visto que o processo de busca do método a ser acoplado à mensagem enviada é iniciado a partir da classe a qual está associado o apontador e não a partir da classe do objeto referenciado.

Por exemplo, é impossível, **analisando-se apenas a linha do envio da mensagem** draw no código da figura 2.20, saber qual das implementações de draw será executada.

[21] do inglês *overload*.

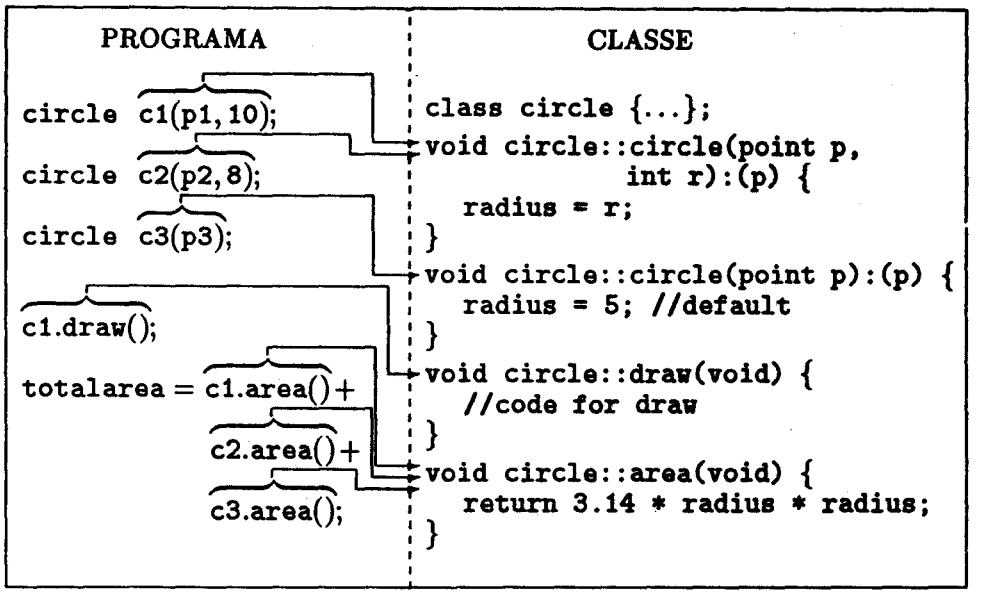

Figura 2.19: Envio e acoplamento de mensagens, na forma **precoce**, em C++

```
point ponto1(8,4);
point ponto2(10,5);
int raio = 10;
int lado = 5;
shape *fig;
...
fig = new circle(ponto1,raio);
fig->draw();
...
fig = new square(ponto2,lado);
fig->draw();
```

Figura 2.20: Acoplamento tardio (*late binding*)

```
class date {
   int day, month, year;
public:
   date();
   date(int d=18,int m=12,int y=1991);
   date(char *datestring);
   date(date &);
};

date today;                    //executa date();
date also_today(today);        //executa date(date &)
date today_too = today;        //executa date(date &)
date long_ago(1,1,450);        //executa date(int,int,int)
date far_ahead("7/8/2100");    //executa date(char *)
date other_date(20);           //executa date(int,int,int)
```

Figura 2.21: Especificação e instanciação de objetos da classe date

2.4.9 Sobrecarga de funções

Um requisito considerado importante em linguagens que suportam o estilo de programação orientado a objetos é o de permitir, na especificação e implementação de classes, que dois ou mais métodos possam ter o **mesmo seletor**. A esta facilidade dá-se o nome de sobrecarga[22] de métodos ou funções. Em termos de suporte a esta característica, o que realmente se verifica é que o compilador, que em linguagens convencionais considerava apenas o identificador da função para identificar o endereço do código a ser executado por ocasião da chamada da função, agora terá que fazer uma análise **sobre os argumentos** desta mensagem.

Na figura 2.21 é mostrada a especificação da classe date em C++. Uma observação sobre este exemplo é a de que, em C++, quando um método possui o mesmo identificador da classe, este recebe a denominação de **construtor** e o código correspondente será executado sempre que um objeto desta classe for instanciado. Na figura 2.21 definimos quatro construtores para a classe date (todos eles com argumentos distintos). Existem também, em C++, métodos que possuem o mesmo identificador da classe, só que precedido de um til (~). A estes métodos (sem argumentos - portanto só existe no máximo um para cada classe) dá-se o nome de **destrutores**, e o seu código será executado sempre que o objeto sair de escopo, ou que ele seja explicitamente eliminado.

[22] do inglês *overload*.

2.4.10 O princípio reativo

Como dito no início do capítulo, uma aplicação desenvolvida sob o paradigma de objetos nada mais é do que um conjunto de classes inter-relacionadas, onde objetos instanciados a partir destas classes trocam mensagens entre si. Estes objetos formadores da aplicação foram "definidos" como sendo uma espécie de "ente vivo", pois possuíam **estado e comportamento** próprios. Dada a necessidade de os usuários terem que acompanhar a execução de uma aplicação e da necessidade de se interagir com estes objetos em aplicações como a mencionada anteriormente é que surgiu uma espécie de modelo de acompanhamento de objetos que se baseia e se vale de duas necessidades **intrínsecas aos mesmos:**

- de eles serem **visíveis** - neste caso o usuário precisa acompanhar e tomar conhecimento do seu estado interno (e modificações sobre ele);
- de eles serem **estimuláveis** - neste caso o usuário necessita atuar sobre o seu estado para que ocorram mudanças na sua representação.

Este modelo de interação e acompanhamento de execução de aplicações recebe o nome de **princípio reativo.** A mudança no estado interno de um objeto pode provocar que outros objetos desta aplicação tenham que "se adaptar" a nova situação, através da **adequação** de seus atributos.

Um exemplo deste modelo poderia ser como o mostrado na figura 2.22, onde existem diversos objetos (`Meter`, `DigiMeter`, `Dial`, `BarChart`, etc.) representando uma aplicação industrial qualquer. Uma observação importante sobre o exemplo, é a de que a forma sob a qual estes objetos são mostrados na tela de um computador não tem relação direta com a forma com a qual estão representados internamente.

O objeto `Meter`, por exemplo, está sendo representado por um mostrador analógico mas, internamente (neste modelo), as únicas informações consideradas essenciais são o seu valor corrente e o intervalo de validade de seus valores. No caso particular estes valores são 38 para o valor corrente e 0-220 para o intervalo de validade.

A modificação dos valores internos através destas representações externas faz com que o objeto, sendo **estimulado**, reaja (modificando inclusive a sua apresentação na tela) e modifique o seu estado interno (caso a modificação na apresentação externa induzida pelo usuário seja pertinente). Depois, esta modificação deve ser propagada aos outros objetos da aplicação que possuam alguma relação com o objeto sendo modificado, promovendo uma espécie de **reajuste geral** do modelo.

Se o objeto `Meter`, do exemplo da figura 2.22, estivesse representando a **temperatura** de um forno industrial (*i.e.* atualmente em 38 graus de temperatura, como mostrado na representação), e o usuário interagisse com a mesma modificando-a para 80^{o}, então, provavelmente outros objetos desta aplicação teriam que reajustar seus valores (por exemplo, o objeto que representasse a **pressão** dentro do forno teria que alterar a sua apresentação e o seu valor interno) para refletir a nova situação.

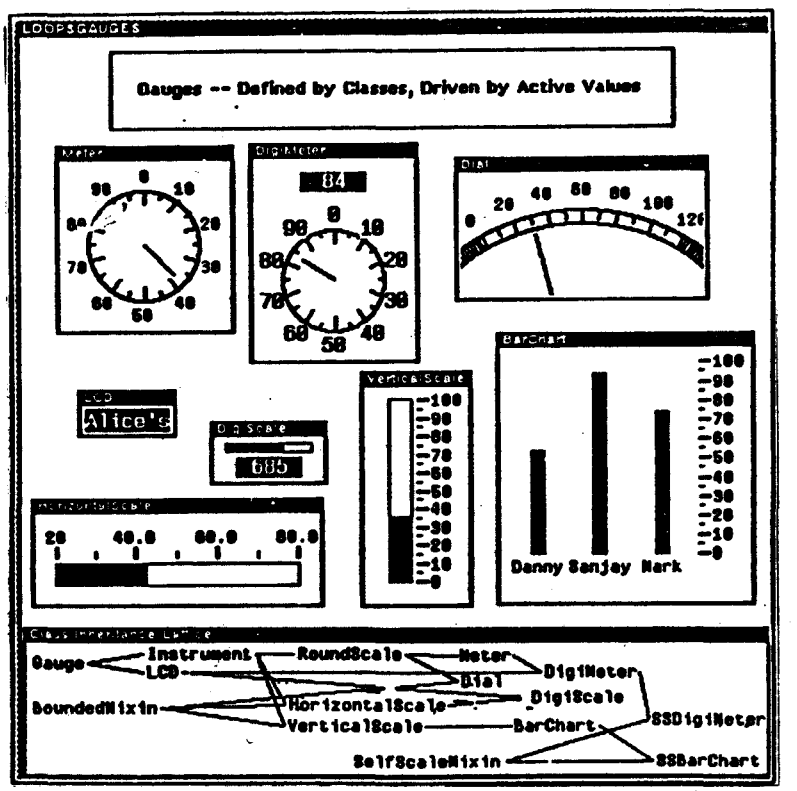

Figura 2.22: O princípio reativo para uma aplicação industrial

O princípio reativo é considerado uma característica desejável no paradigma de objetos e é o **tema central desta dissertação.** A sua definição será estendida nos próximos capítulos, de molde a incluir a definição de múltiplas visões. O princípio reativo para valores numéricos foi implementado sobre o XView (XWindow-System-based Visual/Integrated Environment for Workstations) [Hel90]. A sua implementação consiste, dentre outras coisas, na criação de diversas classes de representação de visões de objetos e a incorporação das mesmas em uma hierarquia de classes bem como na elaboração de um modelo operacional (comunicação entre as visões, mecanismos de atualização de valores, etc.).

Capítulo 3

XView – O Toolkit Adotado

3.1 Introdução

Este capítulo introduz, de forma bastante superficial, o software adotado na implementação do princípio de múltiplas visões com reatividade, descrito no capítulo 4. A idéia central é a familiarização com alguns termos comuns no XView/XWindows citados na descrição e na implementação do problema proposto.

O XView (XWindow-System-based Visual/Integrated Environment for Workstations) é um *toolkit* utilizado para desenvolvimento de interfaces homem-máquina que foi construído e desenvolvido sob o sistema XWindows. Isto implica em dizer que uma aplicação desenvolvida sobre o XView pode fazer uso das primitivas disponíveis no sistema X (Xlib), além de manipular peças de construção de interfaces projetadas e implementadas para este fim (*i.e. menus, canvas, panels, frames, scrollbars, ...*). A criação e manipulação destes objetos é bastante simples, tal como será visto nas seções seguintes deste capítulo.

A funcionabilidade (e apresentação) dos objetos definidos e implementados no XView seguiram o padrão de especificação do OPENLOOK GUI (Graphical User Interface)[1]. O principal fato inovador na adoção do OPENLOOK GUI é o de que ele permite a criação de aplicações através de uma interface consistente e de fácil compreensão por parte dos usuários. O XView foi desenvolvido por um consórcio entre a Sun Microsystems e a AT&T para ser uma espécie de interface padrão para o Unix System V, release 4.

Uma vantagem significativa na adoção do XWindows como o sistema base para construção do XView é a de que a construção do sistema X foi baseada em um protocolo padrão para operações em rede. A adoção desta técnica faz com que aplicações baseadas em X possam ser portadas para diferentes arquiteturas de computadores, pois, ao invés do nível base do XWindows ser construído a partir de *system calls* e de comandos específicos para uma determinada arquitetura, ele foi desenvolvido a partir de uma camada de primitivas acessíveis através de um protocolo padrão localizado entre o sistema operacional e as

[1] OPENLOOK é marca registrada da AT&T.

```
typedef void *Handle;
...
Handle h;
h = WNewComponentSet(myType,myName);

extern "C" {
   Handle WNewComponentSet(cpt_set_type t,cpt_name_t n)
   {
      TComponentSet *s;
      s = new TComponentSet(t,n);
      return (Handle)s;
   }
}
```

Figura 3.1: Interface entre programas escritos em C++ com programas es escritos em C

primitivas do sistema.

A linguagem nativa para construção de programas e aplicações no XView é a linguagem C. A implementação do exemplo proposto, entretanto, foi baseada em C++. Para que esta migração pudesse acontecer sem problemas, algumas modificações superficiais tiveram que ser realizadas no XView (ou nos programas em C) para que o mesmo pudesse se comunicar com o C++: uma aplicação escrita em C pode **criar, manipular** e **visualizar** objetos em C++ através da criação de variáveis ponteiros com tipo opaco (definidas com `void *`) sendo efetuada a associação com os objetos definidos no código em C++ em funções declaradas como "externas" escritas em C. A ilustração de como isto foi implementado é mostrada na figura 3.1, onde um construtor do C++ pôde ser chamado dentro de uma rotina escrita em C.

A outra alternativa para construção de aplicações orientadas a objeto dentro do XView seria a construção de todos os programas em C++ e, para compilação dos módulos, apenas o pré-processador do C++ seria executado; daí para frente, todo o sistema seria compilado com o compilador C padrão e, no processo de ligação seriam usadas, além das bibliotecas do XView e do XWindows, as do C++.

3.2 Estrutura do XView

O princípio básico na construção de um *toolkit* é o de permitir ao programador a construção de uma aplicação através de componentes pré-definidos que possam ser reutilizados e personalizados[2] para formação de uma interface rápida, segura, eficiente, consistente e

[2]ver também seção 2.4.

flexível. O XView é formado por uma coleção de aproximadamente 60 componentes pré-definidos que incluem:

- *frames* - uma área que contém os componentes da interface (*i.e canvas, panel,* etc) e que permite a comunicação entre a aplicação e o gerenciador de janelas (*i.e.* OpenWindows, XWindows);
- *canvas* - onde os programas (desenhos e textos) podem ser desenvolvidos e executados;
- *panels* - uma espécie de painel que contém *menus*, botões, listas com múltiplas escolhas, *scroll lists*, etc;
- *menus* - um componente do *panel* que permite a seleção ou a ativação de opções agrupadas sob uma janela.

Alguns dos objetos e componentes utilizados no XView, bem como a forma de criação e manuseio dos mesmos, serão descritos nas seções seguintes deste capítulo.

O XView (assim como o XWindows) "funciona" à base de atendimento de eventos passados pelo *server*: uma espécie de processo que interpreta e distribui requisições do usuário para os diferentes *clients* ativos no momento. Os eventos no XWindows são gerados a partir das seguintes fontes:

- da entrada do usuário (*click* de *mouse*, entradas via teclado, etc.);
- do gerenciador de janelas (movimentação ou redimensionamento de uma janela);
- do sistema operacional (interrupções, etc);
- de aplicações em geral (mudança de posição de menus, etc).

Os programas desenvolvidos sob o XView recebem eventos de um módulo chamado *notifier*, que funciona como uma espécie de distribuidor de tarefas[3]. A estrutura de programas desenvolvidos sob o XView deve possuir, desta forma, uma rotina que gerencia eventos (uma espécie de *event handler*) e os trata de acordo com o seu tipo. Muitos dos eventos gerados para as aplicações no XView são enviados diretamente para os objetos ou componentes criados na confecção da aplicação, tal como será visto. Os demais eventos devem ser tratados nesta rotina através de uma estrutura de controle do tipo case, onde cada entrada representa o tipo do evento a ser tratado. O XView possui um mecanismo que permite "filtrar" os eventos passados pelo *notifier*. Ou seja, é possível "informar" ao *notifier* quais os eventos que devem ou não ser considerados para uma determinada aplicação. Por exemplo, o usuário pode especificar no XView que apenas eventos de *mouse*

[3]do inglês *event dispatcher*.

devem ser considerados. A partir deste ponto, mesmo que o usuário gere eventos que não estejam relacionados com o *mouse* (como por exemplo o acionamento de uma tecla), o "filtro" contido no XView passa apenas para a aplicação o que foi solicitado. Ou seja, ele ignora este evento.

Uma consideração importante no esquema que o XView introduz é que a maioria dos objetos definidos na construção da aplicação possuem e trabalham através de procedimentos chamados de *callback functions*. Isto vale dizer que a seleção de objetos deste tipo faz com que o código definido pela *callback function* daquele objeto será executado automaticamente. O uso do esquema contido nas *callback functions* faz com que os possíveis eventos, que seriam gerados pelo *notifier* e transferidos para o *event handler* do XView, passem a ser tratados diretamente por estas funções.

3.3 OPENLOOK GUI

Uma característica importante do XView Toolkit é que ele implementa o padrão OPENLOOK para geração de interfaces usuário.

O projeto e implementação do OPENLOOK levou em consideração três princípios básicos para o seu uso:

- simplicidade;
- consistência e
- eficiência.

Devido a estas características, a base da interface do OPENLOOK possui um pequeno **conjunto** de peças através das quais os componentes finais foram montados.

A seleção e o acesso à maioria dos componentes definidos e implementados pelo OPENLOOK é feita através de um cursor que se move por toda a tela e cujas operações (seleção, ajuste e movimentação) ficam a cargo do dispositivo de apontar (*mouse*).

O ambiente implementado baseado no OPENLOOK é composto por uma série de componentes tais como janelas, *menus*, ícones, *frames*, dentre outros. Um exemplo de um ambiente produzido com base no padrão OPENLOOK, é mostrado na figura 3.2.

As janelas do OPENLOOK representam áreas limitadas de interação onde tanto gráficos como textos podem ser exibidos ao usuário.

Os ícones são simbolizados por pequenos quadrados que representam aplicações que não estão habilitadas para interação no momento. O desenho utilizado para descrever um ícone tenta captar o significado associado à aplicação que representa. Na figura 3.2, por exemplo, existem dois ícones no lado direito da tela: um deles consiste de uma representação de uma **caixa de correio** aberta. Isto leva o usuário a associar este ícone a uma ferramenta de manipulação de **correspondências eletrônicas** e que uma nova correspondencia acaba

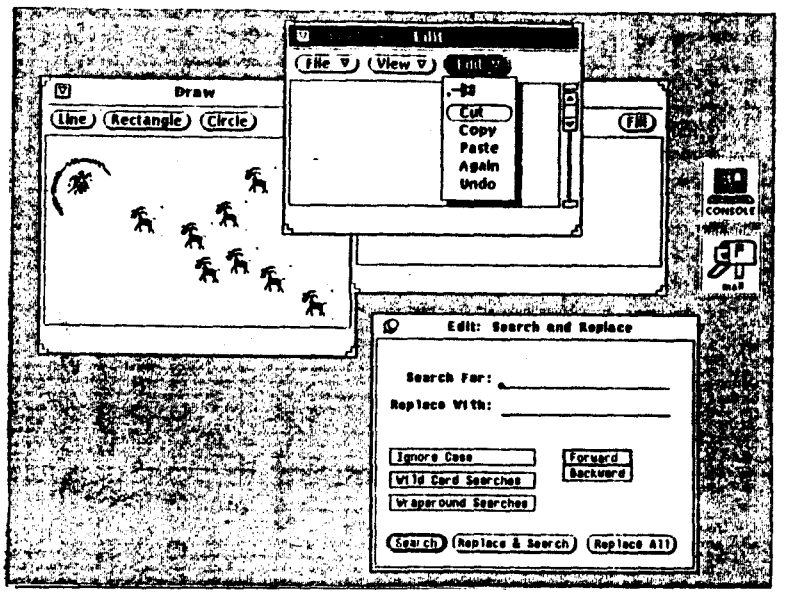

Figura 3.2: Exemplo de um ambiente especificado segundo o padrão OPENLOOK

de chegar - ou está disponível). Existem áreas de estudos específicas para o projeto e apresentação adequada de ícones, conforme dito na seção 1.4.

Um outro tipo de componente definido pelo OPENLOOK e ilustrado na figura 3.2 são os *menus*. Este componente pode ser dividido em, pelo menos, dois tipos: o *pulldown* e o *popup*. Os *menus pulldown* estão disponíveis através do acionamento de botões definidos na área de controle (*panel*) de uma janela e os *menus popup*, que aparecem e desaparecem na posição do cursor, estão disponíveis através da ativação dos botões do *mouse*.

A figura 3.3 ilustra alguns dos componentes idealizados no padrão OPENLOOK. As aplicações, como mostrado, podem utilizar-se de *scrollbars*, *popup windows*, *frames* (de base e de comandos), *scroll lists*, *canvas*, etc.

O objetivo central na definição de componentes do OPENLOOK foi a simplicidade de interação do usuário com os mesmos. Alguns aspectos visuais foram considerados para que a simplicidade procurada pudesse ser implementada:

- desenhos tridimensionais de botões, *menus*, ícones;
- padronização nas cores e fontes utilizados nos *menus*;
- apresentação uniforme dos botões com *status* de selecionado, normal, inativo e ocupado;
- espaçamento padrão e fixo entre as opções de um *menu popup* ou *pulldown*;

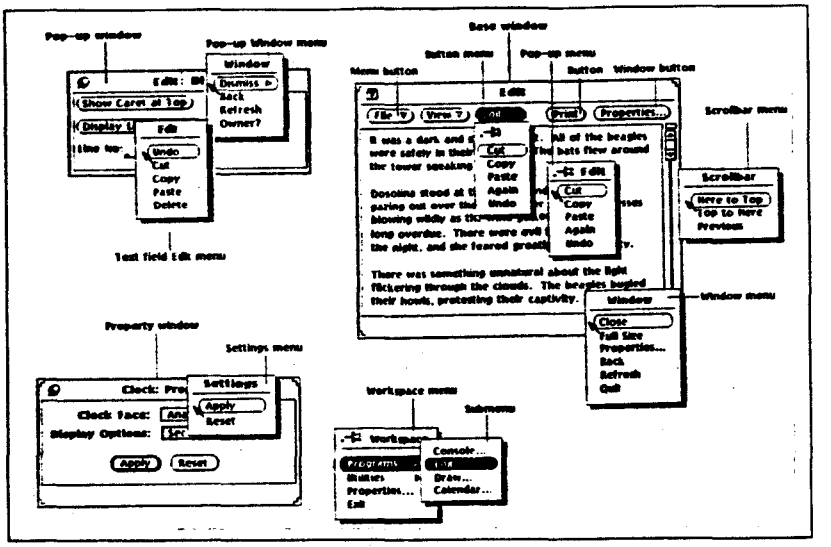

Figura 3.3: Exemplo de componentes idealizados e definidos pelo padrão OPENLOOK

- uniformidade de operação do *mouse* (*i.e. double click*, *select click*, *drag*, *adjust click*), dentre outras.

A utilização do padrão OPENLOOK está incorporada à implementação do XView e a sua maior característica é a de ter viabilizado a sua implementação através de um *toolkit* consistente e de fácil utilização.

3.4 Hierarquia de abstrações do XView

A especificação dos componentes implementados pelo XView está organizada segundo uma estrutura de árvore de objetos. O significado da árvore de especificações definida pelo XView é análogo ao das hierarquias de abstrações mostradas no capítulo 2[4]. Cada nó desta hierarquia especifica e acrescenta uma nova **classe** às já existentes (no XView o termo **classe** também é usado para designar um pacote - *package*). A hierarquia implementada no *toolkit* possui mais de uma **raiz**. A principal hierarquia (ou pelo menos a maior) do XView é a que inclui os componentes interativos propriamente ditos (também chamados de componentes visuais). Dentre os componentes visuais implementados estão o *frame*, *window*, *menu*, *icon*, etc. Esta hierarquia também é chamada de **hierarquia de objetos** e uma parte dela está ilustrada na figura 3.4. O pacote mais abstrato desta hierarquia é,

[4] ver figura 2.7.

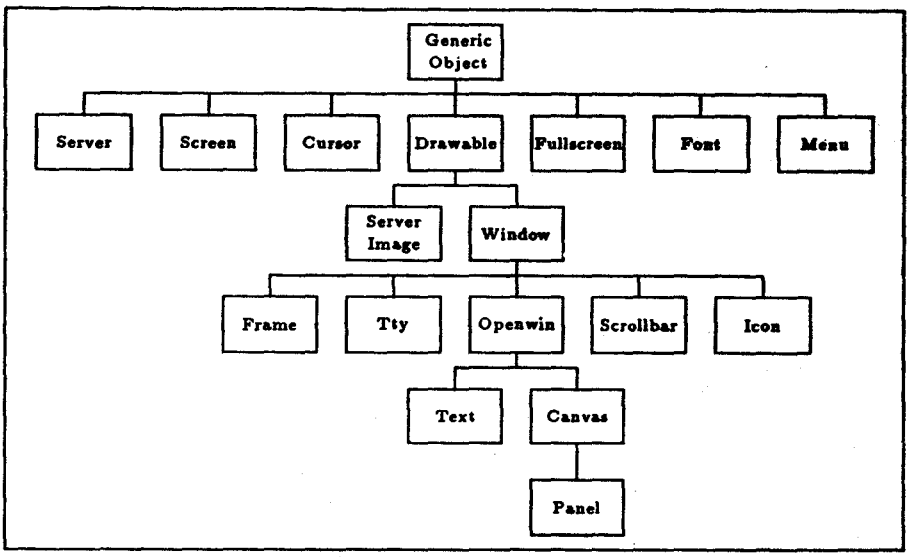

Figura 3.4: Uma hierarquia de abstrações de classes (ou pacotes) do XView (extraída de [Hel90])

como mostrado, o `GenericObject` e é nele que são definidas as propriedades mais abstratas dos objetos do XView. É na classe `GenericObject` que estão definidas e implementadas funções de baixo nível, métodos com comportamento abstratos e os atributos comuns a todos os objetos da árvore. Todos os pacotes definidos nesta hierarquia herdam os atributos e os comportamentos definidos neste objeto e, além disso, acrescentam novas propriedades às já existentes.

Um exemplo deste mecanismo de herança é a classe `Icon`. Na hierarquia da figura ela está localizada como uma subclasse de `Window`, o que parece conveniente, uma vez que um ícone está representado na tela por um pequeno quadrado (geralmente de 64x64 *pixels*), que possui tamanho, largura, profundidade, cores de fundo e de frente, etc: ou seja, todos os atributos de uma janela (*window*) qualquer.

3.5 Criação e manuseio de objetos

Todos os objetos utilizados por aplicações desenvolvidas sobre o XView possuem uma espécie de identificador que caracteriza o componente com o qual se está interagindo. Qualquer operação[5] que se aplique sobre um objeto qualquer requer a utilização deste identificador.

[5]obter informações sobre o seu estado interno ou sobre o valor de um determinado atributo, etc.

A criação e manipulação de objetos se dá através da utilização de apenas seis funções ou "métodos":

- `xv_init` - estabelece um canal de comunicação com o *server*, inicializa o *notifier* e faz consultas a atributos definidos pelo *toolkit* antes da utilização e criação dos objetos da aplicação;
- `xv_create` - permite a criação de um objeto propriamente dito. Tem número de parâmetros variável e retorna sempre um **apontador** para o objeto criado;
- `xv_destroy` - remove o objeto apontado pelo parâmetro. Uma observação importante na destruição de um objeto é a de que a remoção de um objeto irá fazer com que todos os objetos agregados ao objeto sendo destruído, sejam também removidos;
- `xv_find` - procura um objeto, dentre os já criados que satisfaça certas condições passadas como parâmetro. Caso o `xv_find` não encontre um objeto de acordo com as condições estabelecidas, então ele se comportará daí para frente como o `xv_create`;
- `xv_set` - modifica o valor de um atributo (ou de vários atributos) passado como parâmetro;
- `xv_get` - recupera o valor de um atributo especificado pelo usuário na chamada da função.

A criação de alguns objetos para uma aplicação real (o simulador de circuitos implementado nesta dissertação) será mostrada na seção seguinte deste capítulo.

3.6 Uma aplicação típica

Um dos exemplos através do qual o princípio reativo foi ilustrado é um **simulador de circuitos lógicos**, cuja descrição está presente no apêndice B desta dissertação. O princípio reativo, neste exemplo, está incorporado ao simulador na ocasião em que o usuário vai manipular o relógio do simulador.

A figura 3.5 ilustra o *frame* principal do simulador, bem como todos os objetos interativos (botões, *menus*, *panels*, *panel choices*, *canvas*, etc.) presentes no mesmo.

A aplicação ilustrada é tida como uma aplicação típica de desenhos (*draw programs*) e como tal, possui basicamente uma área de *paint* (*canvas*) com os *scrollbars* horizontal e vertical e dois *panels* representando as opções e as portas lógicas a serem inseridas na formação do circuito.

A instanciação de objetos é bastante simples. Alguns dos componentes ilustrados na figura 3.5 estão com o código de sua criação ilustrados nas figuras 3.6, 3.7 e 3.8.

A criação do *frame* da figura 3.5 está ilustrada na figura 3.6. Este componente, como o próprio nome já sugere (`RootFrame`), será o objeto **pai** de todos os demais componentes

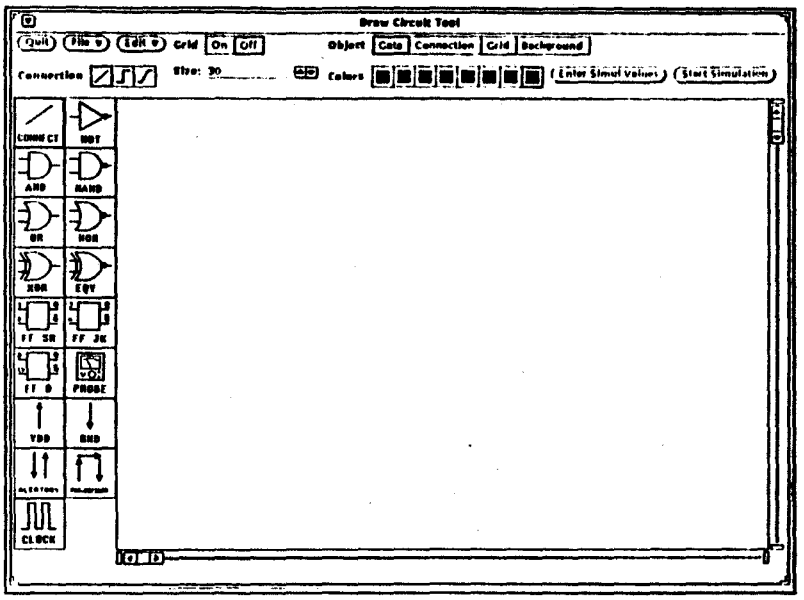

Figura 3.5: Uma aplicação típica desenvolvida no XView

```
RootFrame = (Frame)xv_create(NULL, FRAME,
                    FRAME_LABEL, "Draw Circuit Tool",
                    FRAME_SHOW_HEADER, TRUE,
                    FRAME_SHOW_FOOTER, TRUE,
                    FRAME_LEFT_FOOTER, "",
                    XV_X,              100,
                    XV_Y,              100,
                    XV_WIDTH,          900,
                    XV_HEIGHT,         640,
                    NULL;
```

Figura 3.6: Criação do RootFrame do simulador da figura 3.5

```
GridPanel =
   (Panel_item)xv_create(OptionsPanel, PANEL_CHOICE,
                PANEL_LABEL_STRING,    "Grid",
                PANEL_CHOICE_STRINGS, "On","Off",NULL,
                PANEL_NOTIFY_PROC,     GridProcedure,
                PANEL_VALUE,           1,
                NULL;
```

Figura 3.7: Criação de um dos *panel choices* do simulador

da aplicação, isto é, ele agregará direta ou indiretamente todos os outros objetos do XView criados para a aplicação específica. Isto vale dizer que a destruição deste objeto faz com que **todos os objetos** da aplicação sejam automaticamente destruídos. O primeiro parâmetro do método **xv_create** descrito na figura está relacionado com o **dono** do objeto sendo criado[6]. O segundo parâmetro diz respeito ao **nome do pacote** ou ao nome da classe do objeto a ser criado (no caso o pacote FRAME), e os demais pares de comandos dizem respeito aos valores que devem ser associados aos atributos na criação do componente. No exemplo da figura 3.6, oito inicializações serão realizadas por ocasião da criação do RootFrame: o *string* "Draw Circuit Tool", por exemplo, será atribuído ao parâmetro FRAME_LABEL, o valor TRUE será atribuído ao parâmetro FRAME_SHOW_HEADER, etc.

A figura 3.7 ilustra a criação de um componente chamado GridPanel que é instanciado do pacote PANEL_CHOICE. O GridPanel está localizado no primeiro painel de opções (OptionsPanel) da figura 3.5 e representa (como o próprio nome já sugere) um ítem do painel que implementa uma espécie de lista de opções em que o usuário, através do *mouse*, escolhe uma das alternativas disponíveis. Uma observação importante na criação deste objeto é a definição do identificador da *callback function* associada ao componente. Quando o usuário interage com uma das duas opções disponíveis (On e Off), o *notifier* desvia **automaticamente** a gerência da execução para esta função cujo código implementa a ação desejada pelo usuário para a opção selecionada.

A figura 3.8 ilustra operações sobre componentes já criados. A primeira delas representa a atribuição ou mudança no valor de um atributo no objeto GetFilePanel. O uso deste comando implica em dizer que, a partir do ponto da aplicação onde este comando tiver sido inserido, o *panel item* GetFilePanel estará inativo para o usuário (não será possível executar nenhuma operação sobre ele). Isto é feito atribuindo-se ao estado da variável PANEL_INACTIVE o valor TRUE. Na segunda operação, o usuário recupera o valor

[6]no caso específico deste objeto - o pai de todos, o dono é NULL - significando que o componente RootFrame será associado a um dono pré-determinado pelo *toolkit*.

```
int Canvas RootCanvas;
int Panel GetFilePanel;
int ColorIndex = 0;

xv_set(GetFilePanel, PANEL_INACTIVE, TRUE, NULL);
ColorIndex = xv_get(RootCanvas,WIN_BACKGROUND_COLOR);
```

Figura 3.8: Atribuição e recuperação do valor de um atributo no objeto

do índice no *color map segment* onde a cor de fundo[7] do objeto **RootCanvas** está definida. Isto é feito, como ilustrado, através do método **xv_get** aplicado ao objeto **RootCanvas**, em conjunto com o atributo **WIN_BACKGROUND_COLOR**.

[7] *background color* em inglês.

Capítulo 4

O Princípio Reativo

4.1 Introdução

4.1.1 Motivação

A forma e a metodologia de interação com as aplicações têm evoluído, como visto, para cada vez mais se adequarem ou se aproximarem de metáforas definidas e vistas no mundo real. A redução do *"gap"* semântico, descrito no início do capítulo 2, é sentida em qualquer que seja a área da aplicação a ser desenvolvida. Um requisito que "parece ser" indispensável à concepção de uma aplicação computacional é o de que a interação ou solicitações de atualizações[1] possam ser "sentidas" e visualizadas pelo usuário em tempo de interação com o sistema. Além deste requisito, a **forma** mais adequada de apresentação do objeto com o qual o usuário interage mostra-se, também, um requisito cada vez mais importante e essencial no desenvolvimento de aplicações. É inadmissível, apenas para ilustração e constatação deste fato, que sistemas de controle de vias férreas ou aéreas sejam desenvolvidos sem que as alterações emitidas e solicitadas pelo usuário possam ser constatadas em tempo de interação com a aplicação. Sistemas como estes usam, para efetivação da interação, simulações gráficas para representação de situações reais. Esta filosofia de interação também pode ser sentida em áreas como: controle industrial, gerência de redes de computadores, automação de escritórios, gerência de bases de dados, etc. No **mundo de objetos**, em particular, este requisito (metáfora) parece estar "incorporado"[2] à definição das aplicações. A razão disto é o fato de os objetos formadores de uma aplicação qualquer terem uma necessidade quase que "natural" de serem **vistos e estimulados**. Foi descrito no capítulo 2 (subseção 2.4.10) que os objetos poderiam ser "aproximados" ou associados como sendo "entes vivos" pois possuiam **estado** e **comportamento** próprios. Estes objetos, por sua vez, estão em constante **volatilidade** durante a execução de um programa. Para suprir esta "necessidade" de acompanhamento (do comportamento) e

[1] no sentido de *update-requests*.

[2] ou pelo menos associado.

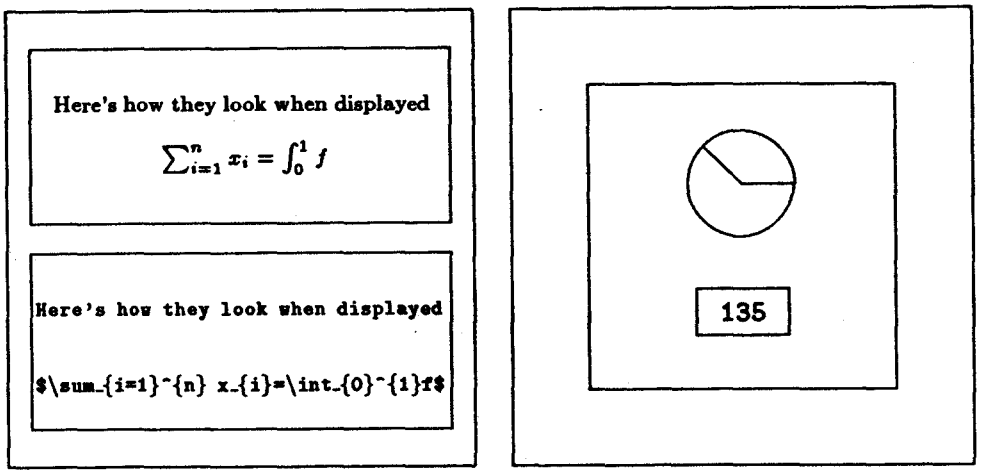

Figura 4.1: Dois tipos de princípio reativo: (a) formatador de textos (b) aplicação numérica.

atualização (do estado) de objetos é que surgiu um modelo de interação que chamamos aqui de **princípio reativo**.

A palavra **reativo**, neste contexto, está relacionada com a atualização de representações ou visões de objetos. Ou seja, a mudança ou atualização do estado interno de um objeto deve provocar uma **reatividade** geral em pelo menos duas etapas: em primeiro estágio na sua visão (ou visões) sendo apresentada ao usuário e em um segundo estágio na aplicação como um todo[3].

A figura 4.1 ilustra o princípio reativo sendo representado em dois tipos de aplicações. Na parte (a) da figura temos um fragmento de código de um formatador de textos em duas visões: a visão formatada e a visão interna. A modificação de valores na visão interna deverá provocar uma atualização na visão formatada. Sem querer entrar em detalhes, este tipo de **reatividade** não é considerada **ainda** completa (ou abrangente) pois a atualização das visões só pode ser feita em um único sentido (da visão interna para a visão formatada).

Na parte (b) da figura é mostrada uma aplicação numérica qualquer em que um **único objeto** é representado e apresentado para o usuário de duas formas diferentes: através de um **mostrador digital** e através de uma **torta gráfica**[4].

Esta forma de interação em que a interação é suportada por dispositivos de interação gráficos como *mouse*, ícones, telas gráficas, janelas, *menus*, etc. está amplamente pro-

[3] atualização ou adeaquação do estado dos objetos relacionados ao objeto sendo modificado.

[4] em inglês *pie chart*.

pagada e é utilizada em estações de trabalhos (ambientes OpenWindows[5], XWindows[6], SunView[7], HPView[8], etc.) e em computadores pessoais (MS-WINDOWS[9]). De uma forma geral, a utilização do princípio reativo e da manipulação direta em interações com aplicações enumera diversas vantagens sobre a sua não utilização, dentre as quais:

- forte **manipulação táctil** dos objetos representados;
- facilidade de aprendizado e uso;
- interação rápida e segura;
- diminuição de escolhas erradas (como será visto);
- manipulação gráfica ao invés de textual, possibilitando a percepção humana em mais dimensões;
- metáfora *desktop*, dentre outras.

4.1.2 Pequeno histórico

O surgimento da filosofia *toolbox* no desenvolvimento de sistemas, como mostrado no capítulo 1 (seção 1.7.1), trouxe uma facilidade enorme na implementação do projeto de interfaces: ofereciam uma variedade imensa de opções para construção dos objetos interativos. A "única" tarefa do programador era a de achar "as peças" adequadas para a montagem da interface pretendida. Este modelo não vingou, pois constatou-se que, devido à grande variedade de opções, associada com a "tendência" (na época) à construção de "interfaces modais"[10], os sistemas e conseqüentemente as suas interfaces proviam facilidades de interação cada vez mais distintas umas das outras. Pesquisas nesta área, que surgiram e se sucederam com a filosofia *toolbox* [B+91], enumeravam pelo menos três princípios essenciais em interfaces e interação (e conseqüentemente no projeto do princípio reativo) em computadores: o forte uso de manipulação direta, a completa eliminação de modos de interação e a tendência da consolidação da filosofia WYSIWYG, citada no capítulo 1 (seção 1.3).

[5] OpenWindows é marca registrada da Sun MicroSystems e da AT&T Corporation.
[6] XWindow System é marca registrada do MIT - Massachusetts Institute of Technology.
[7] SunView é marca registrada da Sun Microsystems, Inc.
[8] HPView é marca registrada da Hewlett Packard, Inc.
[9] MS-WINDOWS é marca registrada da Microsoft, Inc.
[10] interfaces em que a sua utilização estava vinculada à seleção de modos: modo de comando, modo de inserção, modo de alteração, etc.

4.1.3 Eliminação de modos

A criação de modos de operação em aplicações se dava à medida que cada vez mais informações tinham que ser apresentadas para o usuário. Este, por sua vez, se via às voltas com a tela repleta de informações para visualizar, assimilar e depois interagir. A criação de modos passou a ser uma tarefa crucial no desenvolvimento de sistemas, pois a divisão de tarefas em categorias havia-se tornado inevitável.

O princípio da eliminação de modos em aplicações está intimamente relacionado com a idéia de divisão (ou partição) da tela do computador em diversas áreas sobrepostas, também chamadas de janelas (ou *windows*), cada uma representando um contexto distinto de interação.

Esta idéia surgiu e foi implementada inicialmente em ambientes Smalltalk[11]. Neste ambiente cada janela determina um espaço de interação e o controle (interação corrente, ou janela corrente) passou a ser determinadado pela posição do *mouse*.

4.1.4 Manipulação direta e sistemas WYSIWYG

Manipulação direta e sistemas WYSIWYG (What You See Is What You Get) estão intimamente relacionados com a filosofia central do **princípio reativo**. A manipulação direta descreve as formas de interação do usuário com o computador, dado que o mesmo possui o controle direto sobre o que está sendo feito e apresentado em sua tela. O usuário geralmente interage com a aplicação, **diretamente** através de uma das janelas, usando para isto um *mouse* ou um dispositivo de apontar (caneta ótica, cursor, etc.). Modificações sobre as visões externas são transmitidas imediatamente para a aplicação[12] e retransmitidas através de readequações pertinentes das visões da aplicação. Sistemas WYSIWYG estão cada vez mais ligados à manipulação direta, uma vez que a base destes sistemas reside no fato de que o que o usuário está manipulando nas visões será a imagem do produto final. Isto implica em dizer que a atualização dos valores das visões será imediatamente transmitida ao controle da aplicação, para que esta modificação reflita e seja propagada ao núcleo do sistema. Este tipo de esquema é aplicado principalmente a editores de texto. Nestas aplicações o usuário tem uma visão do documento sendo produzido, da mesma forma como ele irá "ver" na impressora como um produto final. Um cursor ou um *mouse* indica, visualmente, a posição exata onde ocorre a interação dentro do texto. A utilização de interação com manipulação direta e a criação dos sistemas WYSIWYG parecem estar **entrelaçadas** à própria definição e implementação do princípio reativo.

[11] Smalltalk é marca registrada da Xerox Corporation.

[12] será visto nas seções seguintes deste capítulo de que maneira será esquematizada esta passagem de informações entre as visões e a aplicação, passando pelo controle e pela abstração da aplicação.

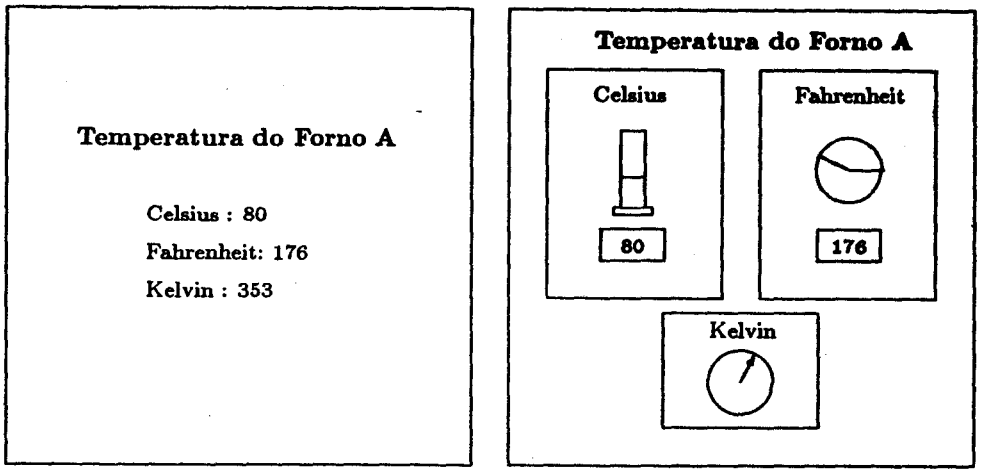

Figura 4.2: Sistemas Reativos de controle de temperatura de um forno industrial: (a) simples (b) com múltiplas visões

4.1.5 Sistemas reativos genéricos

A utilização do princípio de múltiplas visões com reatividade ou simplesmente **princípio reativo** está refletida em um leque imenso de aplicações. Pode-se dizer, em geral, que toda aplicação que requer uma forte interação com o usuário, ou que exija monitorações e acompanhamento de resultados parciais, são aplicaçõs onde a utilização do princípio reativo é adequada e desejável.

Browsers de bases de dados, sistemas de monitoração de resultados, simuladores industriais, configuradores e gerenciadores de redes de computadores, aplicações científicas com forte manipulação de dados, aplicações comerciais, dentre outras, são aplicações típicas de utilização do princípio reativo.

Na figura 4.2 é feita uma comparação entre um sistema reativo simples e um sistema reativo com múltiplas visões. A idéia central do exemplo da figura é mostrar as vantagens de utilização do princípio reativo com múltiplas visões, através da simulação da temperatura de um forno industrial controlado pelo usuário. O sistema produz atualizações nos seus marcadores de controle de temperatura[13] sempre que uma das variáveis sofrer alteração pelo usuário ou por algum agente externo controlado pela aplicação.

As alterações de valores devem ser transmitidas à aplicação e, sempre que houver um risco de hiperaquecimento no forno, o usuário deve ser notificado para que ele possa abrir, por exemplo, um compartimento de escape e a temperatura possa então voltar a baixar.

O funcionamento do sistema é bastante simples. No sistema reativo da figura 4.2 (a), o

[13]em Celsius, Fahrenheit e em Kelvin.

usuário pode interagir com a aplicação e mudar o valor da temperatura do forno, digamos, de 80°C para 100°C. Uma vez finalizada a operação de interação, a mudança é transmitida para o núcleo do sistema, o qual irá reagir aumentando a temperatura do forno (caso seja uma modificação permissível) e propagando o resultado para as demais visões do mesmo objeto (**temperatura** no caso particular). Ou seja, as outras visões representadas na aplicação terão seu estado interno corrigidos para refletir a nova situação: se Celsius = 100 então Fahrenheit = 212 e Kelvin = 373[14]. No sistema reativo da parte (b) da figura, o funcionamento é similar, com as vantagens de que o usuário irá interagir com as visões não textuais de temperatura utilizando um dispositivo de apontar (*mouse*) para operar diretamente sobre as representações gráficas. O sistema irá reagir internamente, da mesma forma que no primeiro caso, só que a reatividade ocorrerá em duas etapas: na primeira, as visões internas (as múltiplas visões) daquele objeto serão atualizadas e, caso a mudança possa ser propagada para os outros objetos, ocorrerá uma mudança em uma segunda etapa[15]. A utilização deste tipo de interação com múltiplas visões gráficas enumera muitas vantagens sobre a sua não utilização:

- a utilização de gráficos para visualização de resultados já **elimina** uma grande quantidade de entradas de valores errados. Ou seja, não é possível subir a temperatura mais do que a dimensionada no termômetro;
- tem-se a idéia clara do escopo de validade da variável (mínimo e máximo);
- pode-se utilizar outras dimensões para mostrar uma **região crítica** de interação. Pode-se usar, por exemplo, a cor **vermelha** para representar a parte da visão onde a interação deve ser mais cautelosa ou até mesmo diminuir a sensibilidade do *mouse* em regiões de interação críticas.
- pode-se emitir **sons** ou mudar a *palette* de cores da aplicação para indicar uma emergência ou uma situação anormal de funcionamento do sistema. Por exemplo, caso ocorra um hiperaquecimento do forno, pode-se mudar a *palette* de cores para indicar que o usuário precisa executar uma intervenção.

A figura 4.3 ilustra um outro sistema reativo. Desta vez é mostrado um software de gerência de redes de computadores. A simples adição de mais terminais ao sistema, ou a modificação do nome de um servidor na visão apresentada ao usuário, deverá propagar-se para as demais visões do sistema (visão interna, visão de disco, etc.), e deverá estar refletida na rede como um todo. Modificações no sistema inverso (da visão textual para a

[14]Celsius = 5 * (Fahrenheit - 32) / 9 e Kelvin = Celsius + 273.

[15]em alguns casos é possível que, mesmo que uma modificação de temperatura em uma visão possa ser transmitida internamente ao objeto, e esteja dentro do escopo de validade da variável, esta modificação não possa ser propagada à aplicação como um todo: o conflito com outras variáveis controladas pelo sistema (*i.e.* o material sendo fundido dentro do forno, etc.).

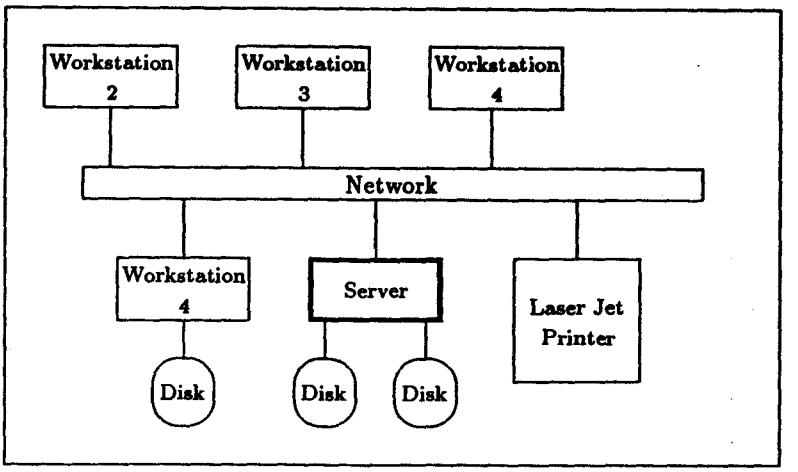

Figura 4.3: Um sistema reativo ilustrado por um software de gerência de redes de computadores

visão gráfica) poderão ser permissíveis[16] e deverão também reagir e propagar as mudanças de forma análoga.

4.2 Adaptação do princípio reativo ao paradigma de objetos

A adoção do princípio reativo em sistemas de computadores mostra que qualquer que seja a solução pretendida para implementação deverá possuir de dois a três níveis de representação de dados: **o nível de visualização externa** ou nível de apresentação final ao usuário (englobando as visões externas, cores, limites, etc.) e **o nível de abstração dos dados** (englobando o comportamento interno e a representação física). Um terceiro nível poderá ser acrescentado ao sistema (ou incorporado ao nível de abstração dos dados), o qual tratará explicitamente do controle das visões e dos dados.

A figura 4.4 ilustra esquematicamente o princípio reativo com seus dois (ou três) níveis de representação de dados e também a **nítida separação que há entre a aplicação e o princípio.**

[16] caso não se esteja usando um equipamento com resolução gráfica de *display* requerida.

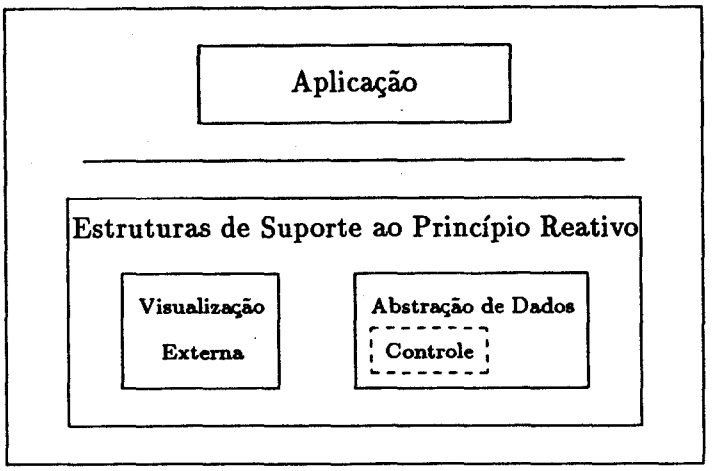

Figura 4.4: Separação entre estruturas de suporte ao princípio reativo e as aplicações propriamente ditas que se utilizam das mesmas

4.2.1 Nível de abstração

Este nível define a separação entre as funções internas da aplicação (o comportamento do sistema) e a visualização do produto. Neste nível, o projetista se concentra na definição e implementação do grau de funcionamento lógico do sistema. O paradigma de orientação a objetos encaixa-se perfeitamente nesta separação entre a abstração e a visualização, uma vez que os seus mecanismos intrínsecos e naturais para a representação de informações abstratas levam a uma especialização até o nível desejado. O nível de controle pode estar residindo em três diferente níveis:

- na própria aplicação;
- no nível de abstração de dados ou
- em um nível novo entre a abstração e a visualização.

A incorporação do controle na aplicação acarreta uma série de problemas. O principal deles é o de que esta decisão, a nível metodológico, faz com que a separação nítida que deveria haver entre a aplicação e as estruturas de suporte ao princípio reativo (independência da aplicação) e entre as funções e as visões externas não fique tão clara. As duas últimas opções mostram-se mais próximas da solução pretendida.

Figura 4.5: Esquema da maneira como sistemas reativos se adaptam a novas situações

4.2.2 Adaptabilidade

Uma outra propriedade decorrente do mecanismo de especialização/generalização de classes em programação orientada a objetos é a que provê **facilidades de adaptação** das estruturas de suporte ao princípio reativo a sistemas computacionais. Além da separação que deve haver entre a **aplicação** e a **abstração-representação-controle** das visões, existe também a necessidade do projetista fazer **ligeiras modificações** na definição original da classe, para melhor adaptá-las à situação pretendida. Um exemplo disto pode ser ilustrado como se, por exemplo, a situação do forno industrial mostrada na seção anterior (figura 4.2) tivesse que ser estendida a uma sala de monitoração, cujos usuários envolvidos ficariam incumbidos apenas de fiscalizar as situações do forno. Nesta sala, o modelo de interação não deve permitir a entrada de dados pelo usuário, uma vez que sua função é apenas a de monitoração e fiscalização dos resultados; então, a **única** modificação que o projetista deve fazer é a criação de uma nova classe, especializada da anterior, eliminando-se a propriedade de entrada de dados. Este novo modelo é ilustrado na figura 4.5 onde a classe mais abstrata representa as visões do modelo com interação e a classe mais especializada representa as visões do usuário na sala de monitoração.

4.2.3 Concorrência

A palavra concorrência, neste texto, em um sentido geral e simples, diz respeito à existência de múltiplas atividades. No princípio reativo, a concorrência se relaciona, como citado no início do capítulo 1 (seção 1.3), a duas facetas:

- múltiplas interações com as visões de objetos;
- múltiplas interações com diferentes aplicações ou diferentes tipos de objetos.

```
void Control :: UpdateViews(int Value)
{
   NumericView *MyNumView = Start();
   while (MyNumView != NULL)
   {
      MyNumView->Draw();
      MyNumView=Next();
   }
}
```

Figura 4.6: Esquema de atualização de visões do nível de controle enfatizando-se o **polimorfismo**

Estes dois tipos de interação tornam-se comuns quando a aplicação se encontra na fase de reajustamento de valores e visões. Nesta fase, múltiplas visões têm que ser rearrumadas para refletir uma nova situação da aplicação. Esta atividade dá idéia de que as visões são atualizadas "concorrentemente". E, qualquer que seja a filosofia de implementação desta propriedade, deve ser feita da maneira mais flexível possível. A definição de **polimorfismo**, mostrada no capítulo 2 (subseção 2.4.7), encaixa-se perfeitamente na visão que o nível de controle deve ter sobre as visões. Em nenhum momento o controle sabe que visões específicas devem ser atualizadas. Ele mantém apenas uma **coleção de visões de objetos genéricos** que fazem parte daquela representação do objeto em questão. Em tempo de execução da aplicação estes objetos genéricos serão associados às visões específicas. Esta propriedade faz com que se possa projetar uma rotina genérica para o **controle**, evitando-se que se tenha um nível de **controle** específico para cada visão o que seria inevitável sem a propriedade do **polimorfismo** entre os objetos. A figura 4.6 ilustra um esquema geral de **uma das atribuições** do controle de estruturas baseadas no princípio reativo em C++ e ilustra um método que poderia ser associado como sendo `UpdateViews`. O uso da propriedade de **polimorfismo** ocorre na linha sublinhada.

4.2.4 Generalidade

Uma característica intrínseca da utilização do princípio reativo é a possibilidade de adequação da sua filosofia a diversos tipos de aplicações. No início deste capítulo foi dito que o princípio reativo nada mais é do que as diversas formas de se interagir e manipular com os objetos de uma aplicação[17], com a propriedade de se "encaixar" ou se adequar às

[17] passando pelos níveis de abstração, representação e visualização.

aplicações com forte interação e manipulação de dados.

A propriedade de **generalidade** em relação à programação orientada a objetos funciona de forma natural, pois a própria definição de abstração de dados, mesclada às propriedades de modelagem conceitual de **abstração/especialização**, faz com que os sistemas reativos possam ser traduzidos para plataformas distintas. O caminho para que isto aconteça resume-se na correta modelagem dos dados, ou seja: na definição correta das classes, na correta organização entre estes "tipos" e na utilização devida/apropriada do mecanismo de **herança**.

A utilização das características, além de deixar clara a separação nítida que deve haver entre os níveis de interação desejados[18], faz com que o ambiente projetado e a interface produzida sejam de transporte extremamente simples, fácil de usar e entender.

4.2.5 Vasos comunicantes

A utilização do princípio reativo induz uma extensão bastante intuitiva em relação ao modelo simplificado da reatividade, descrito no início do capítulo (reatividade local). A idéia central desta discussão é a de que um objeto[19] guarda uma relação de **vasos comunicantes** com outros objetos da aplicação. Ou seja: a alteração de uma visão de um objeto provoca um reajuste ou, utilizando a analogia anterior, uma **comunicação através de vasos**, como a descrita na figura 4.2 do início do capítulo, e idealizada segundo a figura 4.7. A "relação de vasos comunicantes" ilustrada nesta figura mostra que, apesar do êmbolo ter tido um deslocamento d_1 maior do que o deslocamento oposto d_2, a relação entre estes dois agentes[20] deve permanecer **constante** durante toda a fase de interação do usuário com o sistema. Ou seja, o volume deslocado pelo êmbolo do lado direito deve ser igual ao volume deslocado do lado esquerdo:

$$v = d_1 * l_1 * p_1 = d_2 * l_2 * p_2$$

onde: v - *volume*
d - *deslocamento*
l - *largura*
p - *profundidade*

[18] visualização, abstração, ...

[19] e suas representações ou visões externas.

[20] o deslocamento **provocado** pelo êmbolo do lado direito e o deslocamento **agravado** pelo êmbolo do lado esquerdo.

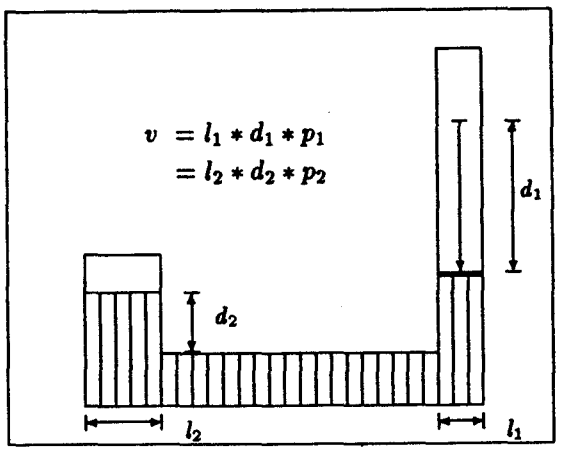

Figura 4.7: Relação de vasos comunicantes

O potencial desta nova extensão faz com que a utilização do princípio reativo seja imenso e o grande benefício do paradigma de objetos é o de ter viabilizado a sua implementação em forma de softwares *toolkits* ou softwares *framework*.

4.3 O modelo MVC do Smalltalk

O modelo MVC não se aplica **especificamente** a uma implementação do princípio reativo. A definição deste modelo pode ser abstraída e aplicada à definição do modelo de interação com o usuário de uma forma geral.

O modelo MVC foi idealizado e implementado inicialmente no ambiente Smalltalk-80. O principal fato inovador da implantação deste modelo foi a separação arquitetural em interfaces orientadas a objetos entre os tão falados aspectos de interação e aspectos de funcionabilidade.

Uma aplicação desenvolvida sob a estrutura MVC, como o próprio nome já sugere, é composta de um **modelo** que, por sua vez, consiste em uma dupla **visão-controlador**.

O **modelo** é a parte que representa a abstração da aplicação e é nela que é implementada a interface para a aplicação. Um **modelo** define uma aplicação propriamente dita, e permite a conexão um par **visão-controlador** como componentes dependentes[21]. Qualquer modificação no estado do objeto será transmitida ao par **visão-controlador** dependente.

O objetivo da **visão** é o de mostrar ao usuário alguns aspectos particulares do objeto (ou da aplicação). O componente de **visão** sempre envia uma mensagem ao **modelo**

[21] para cada visão existirá um controlador.

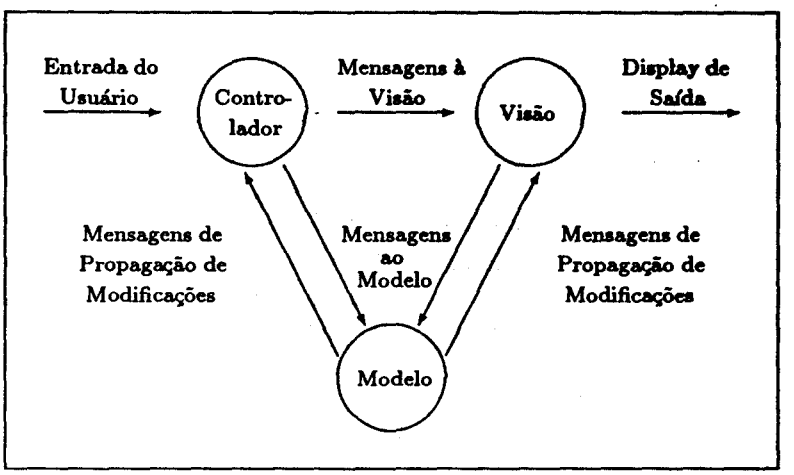

Figura 4.8: O modelo MVC

correspondente/associado, requisitando informações sobre o estado do objeto para uma posterior adaptação. Pode haver várias visões de um mesmo objeto. Neste caso, uma possível mudança no aspecto de uma **visão** provocada pelo **controlador** faz com que uma mensagem seja enviada ao **modelo** que por sua vez, distribui a atualização para todos os seus dependentes, *i.e.* todas as **visões** associadas. O componente **modelo** e o componente **visão** trabalham juntos: o modelo provê às visões informações para atualização dos valores, e as visões proporcionam maneiras de o modelo poder ser visto.

O componente **controlador** executa a coordenação da ação do usuário (por exemplo, via *mouse* ou teclado) com o **modelo** da aplicação e o seu par **visão** associado. O **controlador** interpreta entradas do usuário, em função do contexto de visualização corrente. Por exemplo, o *mouse*, dependendo da região da tela onde ele esteja posicionado, pode selecionar opções de *menu* ou atualizar visões de objetos; para isso, o usuário posiciona-o dentro da visão e atualiza a representação para o novo valor através da manipulação do cursor via *mouse*.

A figura 4.8 ilustra graficamente o modelo MVC e a relação/troca de mensagens entre seus componentes.

O esquema/modelo MVC teve vários sucedâneos em que a questão das múltiplas visões de objetos foram incorporadas ao modelo e vários mecanismos de atualização de visões foram experimentados.

4.4 O modelo PAC

4.4.1 Introdução

O modelo PAC - Presentation-Abstraction-Control é um dos modelos que sucederam ao MVC descrito na seção anterior. Ele foi idealizado para implementação da arquitetura de um sistema de gerência de interface com o usuário, chamado MOUSE, em 1987, na França, por Coutaz [Cou87]. O PAC é um modelo de representação orientado a objetos que pode ser aplicado em qualquer nível de abstração de uma interface homem-máquina.

A idéia central deste modelo[22] é a separação entre os aspectos relacionados com a sintaxe e a semântica de uma interação. O modelo é dividido em três partes (como o próprio nome já sugere):

- o nível de **apresentação** (*presentation*);
- o nível de **abstração** (*abstraction*) e
- o nível de **controle** (*control*).

O nível de apresentação define a sintaxe da interação. É neste nível que é implementado o comportamento externo do objeto (ou da aplicação), ou seja, como ele é percebido pelo usuário. Fazem parte deste nível apenas os atributos particulares de cada visão dos objetos sendo especificados.

O nível de abstração define a semântica da interação com o objeto (ou com a aplicação). É neste nível que é definido o domínio de validade do objeto, o seu valor corrente, e que tipo de ações ele é capaz de executar[23].

O terceiro e último nível é o que trata do **controle**. Este nível funciona como uma espécie de agente intermediário entre a **apresentação** e a **abstração** controlando aspectos de consistência entre o nível de abstração e o nível de visualização. Encarrega-se do disparo de mensagens para estes dois níveis e abstrai-se de aspectos de representação (interna e externa).

A figura 4.9 ilustra o modelo PAC com seus três níveis de representação.

4.4.2 Funcionamento

A apresentação no modelo PAC é representada por um conjunto de classes que ora chamamos de **objetos interativos da aplicação**. Na figura 4.9, por exemplo, o objeto interativo está representado por uma **torta gráfica**. Os objetos interativos implementados para este texto estão descritos no apêndice A desta dissertação.

[22] bem como a do MVC e de muitos outros que o sucederam.

[23] por exemplo: se ele pode ser atualizado pelo usuário, etc.

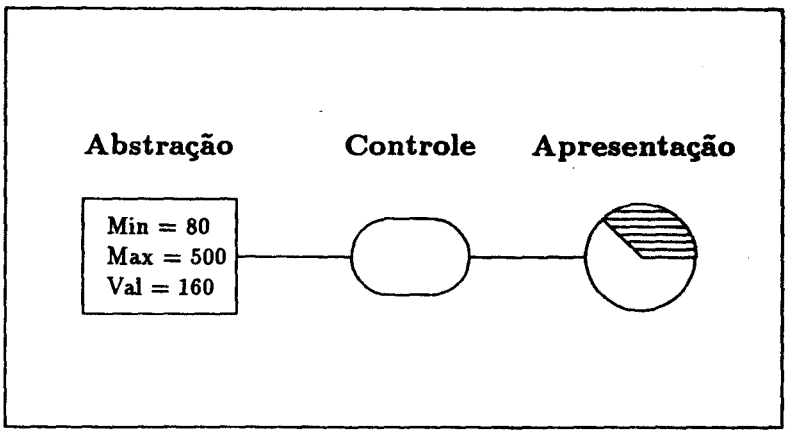

Figura 4.9: O modelo PAC e os três níveis de representação

O funcionamento do modelo PAC simplificado, como mostrado na figura 4.9, bem como alguns atributos de informações contidas em cada nível pode ser resumido para este exemplo, como:

- **apresentação**: raio da torta, o centro da figura, a cor de preenchimento, posição dentro da janela, os limites de interação do usuário (área de entrada de dados), etc.
- **abstração**: um valor inteiro representando o estado corrente do objeto e um escopo de validade contendo um mínimo e um máximo.
- **controle**: consistência e a centralização do modelo. É responsável pelo envio de mensagens para a **apresentação** e para a **abstração** (por exemplo, a responsabilidade do envio das mensagens. Neste exemplo: `UpdateView` para a **apresentação** e `ChangeVal` para a **abstração**).

O modelo de interação é bastante simples. O usuário interage com o nível de **apresentação** do objeto interativo, modificando o seu aspecto externo. Na figura 4.9, por exemplo, se o usuário modificar o tamanho do pedaço da **torta** sendo apresentado, a **apresentação** notifica o **controle** sobre a alteração desejada pelo usuário e este irá provocar uma atualização de valores, tanto na **apresentação**, quanto na **abstração**, através da modificação do valor interno do objeto interativo e da visão externa associada (isto é feito, como será descrito nas seções seguintes deste capítulo, através do envio das mensagens `ChangeVal` e `UpdateView` para a **abstração** e para a **apresentação**, respectivamente). A atualização da **apresentação** pode acontecer também, mesmo que o valor interno não tenha sido modificado pelo usuário. Neste caso, a modificação deverá ter sido provocada por um dos dois (ou ambos) fatores:

- mudança pela aplicação do escopo de validade do objeto. Neste caso, a **apresentação** precisa se "rearrumar" para refletir o novo escopo. Por exemplo, se na figura 4.9 o valor mínimo e máximo do objeto fossem modificados para 0 e 360 respectivamente, o **controle** teria que enviar métodos do tipo `SetMin` e `SetMax` para a **abstração** e métodos do tipo `UpdateView` para a **apresentação**. É interessante notar, neste exemplo, que a mudança de escopo de validade do objeto terá reações específicas para cada nível: a **abstração** só se "interessa" em saber qual é o novo domínio, esquivando-se de detalhes de apresentação. Enquanto que a **apresentação** se "interessa" por detalhes de visualização externa, utilizando-se do domínio de validade apenas para a representação da nova escala visual e para representação do valor interno.

- mudança provocada por algum **agente externo** a este objeto. Neste caso, o **controle** comporta-se como se estivesse sendo feita uma modificação pelo usuário: envia um método chamado `ChangeVal` para a **abstração** e `UpdateView` para a **apresentação**. A natureza desta modificação reside no fato de que o objeto sujeito a sofrer interações está relacionado com outros objetos controlados pela aplicação. A modificação do valor do objeto através de uma interação, provocada sem que tenha ocorrido qualquer ação do usuário, é perfeitamente aceita (ou normal) em sistemas reativos com reatividade composta, como já descrito nas seções anteriores deste capítulo.

4.4.3 Múltiplas visões

Uma aplicação pode, como esperado, ser composta de múltiplos objetos interativos representando uma única entidade de memória. A figura 4.10 ilustra o modelo PAC estendido para suportar múltiplas visões de objetos interativos. O funcionamento deste modelo estendido é análogo ao descrito na seção anterior, entretanto ele introduz características interessantes quanto aos mecanismos de **atualização de visões** e de **controle de diálogos** como será mostrado.

Na figura 4.10 é apresentada uma variável com quatro visões interativas:

- uma torta gráfica;
- um mostrador digital;
- um tubo vertical e
- um *slider* horizontal

O nível de abstração está constituído, neste modelo, pelo valor atual do objeto interativo e pelo escopo de validade (mínimo e máximo). Uma característica interessante, neste nível, é a de que a abstração referente ao **mostrador digital** não possui valores

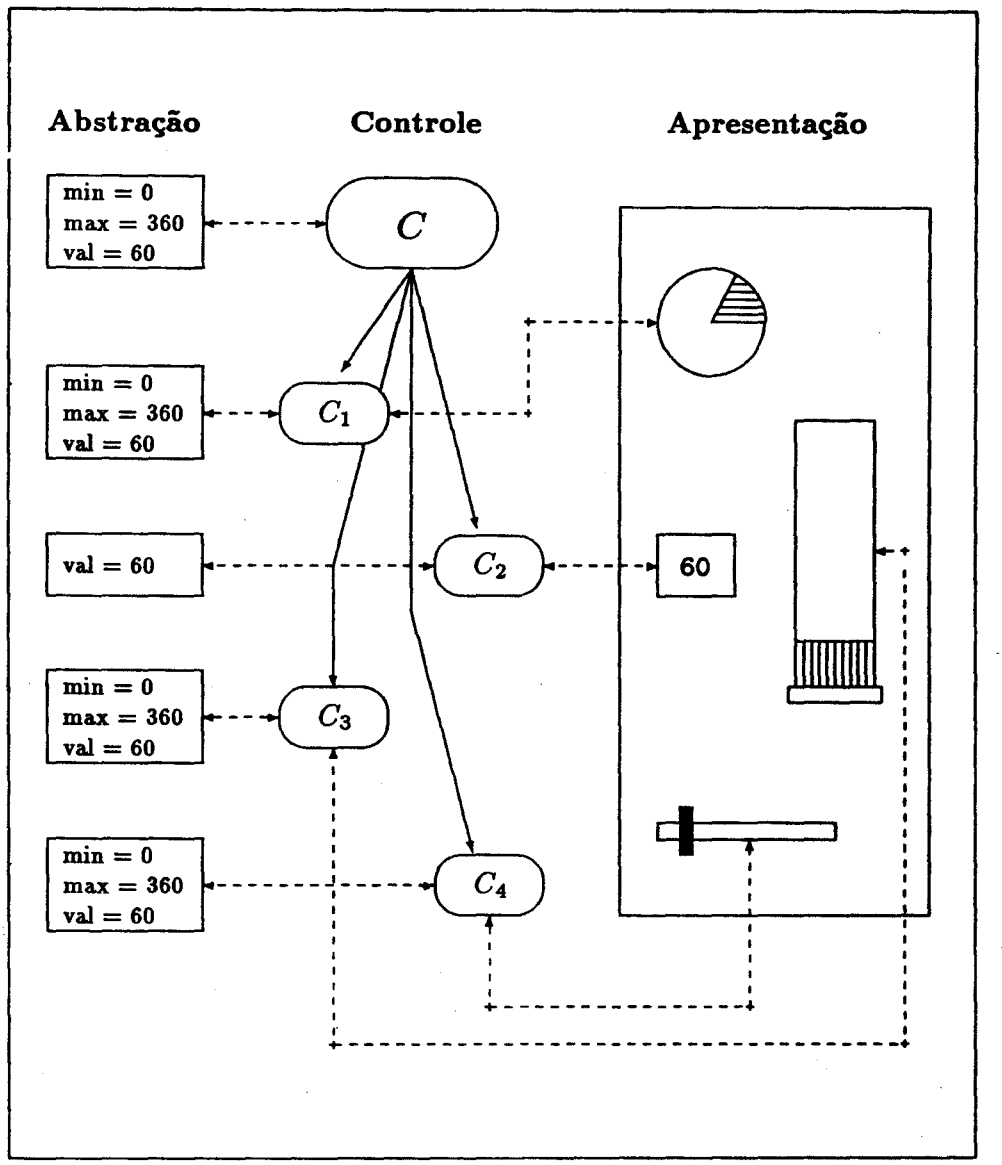

Figura 4.10: O modelo PAC com múltiplas visões

mínimo e máximo. A inexistência do escopo, neste caso, é justificada, pois o objeto interativo **mostrador digital não proporciona** ao usuário uma visualização de limites da variável. O usuário, quando da interação com este objeto, não sabe, considerando apenas este espaço de interação, se o valor por ele inserido, será aceito ou não pela aplicação. E, como a **abstração** deve ser um **reflexo** do que a **apresentação** proporciona ao usuário, a verificação de escopo ficará em outra instância do nível de abstração.

O componente de **controle**, por sua vez, terá uma estrutura similar ao encontrado no modelo simplificado da seção anterior. O controle C será particionado em tantos subcontroles quantos forem os números de representações de objetos interativos. Desta forma, no exemplo da figura 4.10:

- o subcontrole C_1 corresponderá à representação da **torta gráfica**;
- o subcontrole C_2 corresponderá à representação da **mostrador digital**;
- o subcontrole C_3 corresponderá à representação da **tubo vertical**;
- o subcontrole C_4 corresponderá à representação da *slider* **horizontal**;

Existirá também um controle central C que centralizará o disparo de ações para os subcontroles e corresponderá à janela global em que as quatro visões estão representadas.

Para ilustração do funcionamento deste modelo, suponha-se que o usuário deseja modificar o valor da variável de **60** para **100**. Neste caso, ele interage (*via mouse*), com **uma** das representações desta variável, modificando a sua representação. Digamos que a interação foi realizada através da representação da **torta gráfica**. Esta representação reconhece a intenção do usuário e imediatamente notifica ao seu subcontrole (C_1) a tentativa de mudança. O subcontrole C_1, por sua vez, certifica-se com a abstração de que a mudança está dentro dos "limites" permissíveis. Caso a alteração seja válida, então o subcontrole C_1 passa a informação (o novo valor) para o controle central C que irá, então, enviar mensagens em *"broadcast"* para todos os subcontroles, "ordenando" a atualização das **abstrações** particulares e das **visões** correspondentes.

Algumas considerações são interessantes neste esquema de atualização de visões:

- caso o valor solicitado seja rejeitado pelo controle, nenhuma modificação nas visões precisará ser realizada, pois a atualização se dá apenas (inclusive no objeto que solicitou a mudança) **depois** que o controle contral "autorizar" o **reajustamento**;
- mensagens em *"broadcast"* para os subcontroles podem ser entendidas como sendo mensagens **polimórficas** do controle para as visões;
- toda notificação de rejeição passará/partirá do controle central;
- caso o subcontrole sendo analisado não "consiga descobrir" se o valor solicitado está dentro da faixa permitida (por exemplo o subcontrole C_2), ele passa a execução para o controle central que faz a análise;

- é interessante notar que as modificações externas impostas pela aplicação "entram" neste modelo apenas pelo controle central, fazendo com que o modelo de interação permaneça intacto.

4.4.4 Considerações sobre o modelo estendido

Esta abordagem tem duas características interessantes como dito no início da subseção anterior:

- a atualização das visões é centralizada no controle C. A razão disto é a de que os subcontroles não precisam saber sobre "quem são" e "quantas são" as outras visões envolvidas na aplicação. O controle C é que mantém uma espécie de lista de visões ativas e que, portanto, devem ser atualizadas. A adoção deste modelo simplifica bastante a implementação do princípio reativo, uma vez que ele evita introduzir um controle maior em cada representação externa para saber quais os objetos interativos ativos;

- o controle dos diálogos nas visões é particionado. Ou seja: cada objeto interativo possui uma área de interação própria (que é a sua janela). O usuário, entretanto, pode interagir com qualquer uma das visões, sem que isto imponha problemas de controle ao modelo. O controle é feito a nível local (pelos C_i) e, em uma instância maior, pelo controle central (com funções mais genéricas em relação ao modelo).

4.4.5 Interesse pelo modelo PAC

As razões para a escolha do modelo PAC na implementação do princípio reativo são, além das já mencionadas na subseção anterior, as seguintes:

- o modelo define um *"framework"* consistente para construção do princípio reativo. A comunicação entre a aplicação e o modelo é feita de maneira conceitual e não através de detalhes semânticos de baixo nível, irrelevantes para a aplicação;

- introduz um nível de controle ao modelo, servindo de interface entre a abstração e a apresentação e, em nível mais abstrato, entre a aplicação e o modelo;

- deixa clara a utilização dos principais conceitos de programação orientada a objetos:

  - mensagens polimórficas do controle para as visões;
  - herança de classes na representação dos objetos interativos (características comuns são abstraídas em classes projetadas para este fim, ...) e no controle central e particionado;
  - facilidade de customização de objetos interativos através da especialização de classes já existentes, sem alterar visões e abstrações já projetadas.

- o modelo deixa bastante clara a função centralizada do **controle**. Ou seja, uma entidade representada no modelo tem a função específica de enviar mensagens à **abstração** e à **apresentação**, sincronizando os seus comportamentos. No modelo MVC, por exemplo, este controle estava diluído nas três entidades (no **modelo**, na **visão** e no **controlador**);

- o modelo incorpora o comportamento de entrada e saída dos objetos interativos, apenas no componente **apresentação**. A adoção deste princípio faz com que mudanças solicitadas pelo usuário (eventos de entrada) possam ser sentidas imediatamente (eventos de saída) permitindo, por exemplo, o **ajuste fino** de uma visão de forma mais eficiente que no modelo MVC, onde estas ações estão distribuídas nos componentes **visão** (eventos de saída) e **controlador** (eventos de entrada)[24].

[24] a adoção desta técnica tem a vantagem da flexibilidade (pode-se mudar a sintaxe da entrada sem que a saída esteja envolvida), mas isto induz também um certo grau de à "ineficiência" descrito acima.

Capítulo 5

Implementação do Princípio Reativo

5.1 O modelo de objetos

Para criação e implementação do princípio reativo exposto nas seções anteriores, algumas premissas foram tidas como "ponto de partida":

- a solução pretendida deveria ser a mais independente possível do software utilizado (XView);
- seguir orientação segundo o modelo PAC;
- houvesse facilidade de criação/destruição/modificação de objetos interativos por parte da aplicação;
- o nível de **controle** deveria estar em um patamar diferenciado da **abstração** e da **representação** (ou seja, fora da aplicação e da abstração de dados).

5.1.1 Decomposição do problema em objetos

O primeiro fato encontrado para criação do modelo de objetos é a criação dos objetos interativos propriamente ditos. Ou seja, a **identificação** de que objetos são necessários para a implementação do problema, bem como as operações que serão necessárias aplicar sobre os mesmos.

Claramente é necessário representar os objetos interativos em termos de um **conjunto de peças** de construção de interfaces ou **peças de visões externas**, a partir das quais interfaces baseadas no princípio reativo poderão ser construídas.

Uma das principais características a ser explorada, quando se programa no estilo orientado a objetos, é a busca ou a identificação da **comunalidade** entre os objetos formadores

de uma aplicação. Para formar a hierarquia de abstrações pretendida, utilizam-se os conceitos definidos e expostos no capítulo 2.

Em relação ao problema do princípio reativo, identificamos claramente algumas semelhanças entre os objetos interativos (qualquer que seja a sua natureza).

Todo objeto interativo possui como parte de sua memória:

- uma posição dentro da janela de interação (ou seja, uma linha e uma coluna);
- um valor corrente;
- uma área de interação.

Poder-se-ia ter inserido, como parte da memória privada de um objeto interativo genérico, o escopo de validade (mínimo e máximo) do mesmo. Entretanto, esta decisão iria fazer com que a implementação perdesse um pouco da compatibilidade com o modelo PAC. Como visto na seção anterior, cada subcontrole de um objeto interativo comunica-se diretamente com a subabstração correspondente para consulta dos limites de validade do objeto sendo modificado. A inclusão do escopo de validade, neste nível de abstração, faria com que esta consulta não fizesse mais sentido.

Em relação às operações ou ao **comportamento** genérico de um objeto interativo podemos identificar algumas **operações virtuais**[1] (neste sentido podendo significar: sem comportamento claramente definido ou também com comportamento **nulo** ou ainda **com algum** comportamento *default*), comuns aos objetos interativos:

- inicializar e recuperar coordenadas cartesianas do objeto (para criação ou modificação da posição física do objeto na área de interação);
- recuperar o tamanho da área ou o espaço alocado pelo objeto;
- modificar os limites do escopo de validade do objeto[2];
- mostrar e modificar o valor atual do objeto;
- desenhar o valor corrente do objeto nas diferentes formas de representação (operação tipicamente com código **nulo** a nível genérico).

Identificadas as operações básicas e comuns aos objetos interativos, a próxima etapa será a identificação dos objetos interativos específicos. O apêndice A desta dissertação traz uma descrição e especificação de todos os objetos implementados, bem como a descrição dos parâmetros configuráveis.

[1] ver também seção 1.7.

[2] estes métodos, apesar de terem definição nula na abstração dos objetos interativos terão que ter os protótipos aqui definidos para que o seu código correspondente especializado possa ser executado nas subclasses como sendo métodos virtuais ou polimórficos.

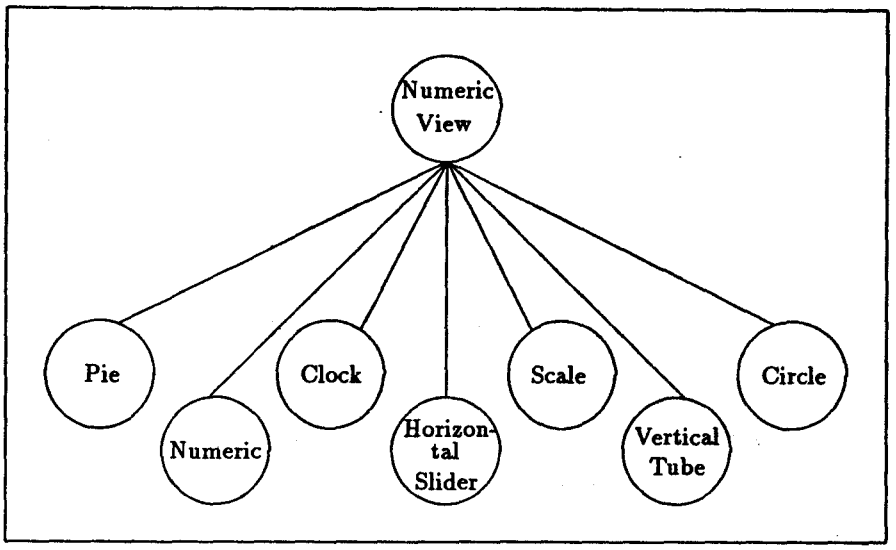

Figura 5.1: Hierarquia de abstrações para alguns objetos interativos

A figura 5.1 ilustra a hierarquia de abstrações definida. Para efeito de simplificação da figura, apenas uma parte dos objetos interativos implementados está representada. A classe `NumericView` corresponde, desta forma, à classe mais abstrata de todas as classes de objetos interativos. Os segmentos de reta correspondem às operações abstratas de **especialização**[3] e de **generalização**[4].

Serão apresentados a seguir alguns aspectos dos objetos interativos definidos bem como alguns de seus atributos. A primeira descrição será a da classe `Pie` e, devido a este fato, uma apresentação mais detalhada será realizada. A representação visual bem como a especificação do comportamento e a definição da memória privada estão, como já dito, ilustrados e representados no apêndice A.

- `Pie` - implementa uma torta circular gráfica em que a área preenchida ou hachureada corresponde ao valor atual do objeto. A figura A.7 do apêndice A ilustra a visão externa implementada deste objeto. Os parâmetros configuráveis são:
  - o valor do raio e
  - o limite de validade do objeto.

O valor corrente do objeto é calculado através da posição selecionada ou apontada

[3] na análise *top-down*.
[4] na análise *bottom-up*.

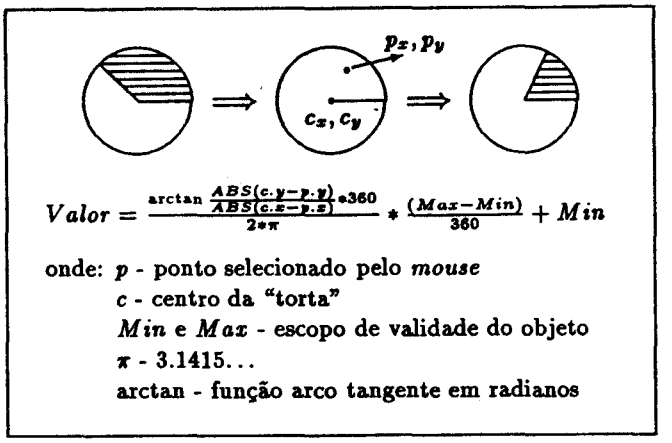

Figura 5.2: Atualização do valor corrente para a classes `Pie`

pelo *mouse*, com auxílio de uma regra de três simples, cujo resultado é mostrado na figura 5.2. As variáveis responsáveis pela mudança de valor no objeto são:

- posição selecionada pelo *mouse* (linha e coluna);
- centro da torta (linha e coluna) e
- valor mínimo e máximo assumido pelo objeto.

- `Clock` - implementa uma espécie de relógio analógico onde o único ponteiro indica sempre o valor corrente do objeto. A visão externa deste objeto é mostrada na figura A.2 do apêndice A e os únicos parâmetros configuráveis (bem como os da classe `Pie`) são:

  - o escopo de validade e
  - o raio do relógio.

- `HorizontalSlider` - implementa um objeto do tipo *slider*[5]. É composto de duas peças interativas: o `SliderBar` e o `SliderHandler`. O seu funcionamento é bastante intuitivo (bem como a de todas as classes do princípio reativo), e o seu valor é atualizado sempre que a posição do `SliderHandler` se alterar.

  A figura A.4 do apêndice A ilustra a visão externa do `HorizontalSlider`, e os seus parâmetros configuráveis são:

[5]ou escorregador, deslizador.

- o seu escopo de validade e
- o tamanho do `SliderBar`.

Uma descrição mais detalhada das classes descritas pode ser encontrada no apêndice A, bem como a especificação e visão externa das demais classes implementadas.

A hierarquia de objetos interativos mostrada na figura 5.1 implementa alguns dos controles previstos no modelo PAC da figura 4.10.

As subabstrações estão representadas pelas memórias privadas de cada classe especializada. A subabstração da classe `Pie` corresponde à memória privada da classe `Pie`, e a subabstração da classe `Numeric` corresponde à memória privada da classe `Numeric`. Um problema ou uma dúvida pode surgir deste esquema inicial: A memória da classe `Numeric` e a de todas as outras classes definidas neste nível da hierarquia não possuem memória do **valor corrente** do objeto! O esclarecimento para esta questão pode ser a de que a memória da subclasse `NumericView` também faz parte da memória da subclasse em questão via mecanismo de herança e, portanto, faz parte da subabstração das classes especializadas.

A hierarquia inicial mostrada na figura 5.1 implementa, como dito, alguns dos controles previstos no modelo PAC: o **nível de abstração** (por enquanto apenas as subabstrações das classes especializadas) que corresponde à memória principal dos objetos interativos; e o **nível de apresentação** que corresponde a **uma parte** do comportamento definido para os objetos especificados.

O **nível de controle** foi definido em um nível diferenciado em relação aos de abstração e de representação. Como dito na seção anterior, um dos principais pontos levados em consideração na elaboração do modelo de objetos foi a separação entre os três níveis envolvidos e a versatilidade do **controle** na criação de objetos.

5.1.2 O nível de controle

A versatilidade na criação de objetos está intimamente relacionada com a possibilidade de criação e expansão das classes formadoras dos objetos interativos, sem que o **controle** sobre elas necessite sofrer mudanças em seu código fonte. Em outras palavras, o código implementado para o controle não precisa saber quais os objetos que com ele estavam se relacionando.

Partindo deste princípio, o **controle** não deveria então enviar mensagens, especificando na chamada o objeto destinatário. Por exemplo, o envio de mensagens tais como:

```
Pie.ShowValue(30); ou
HorizontalSlider.Draw(); ou
Clock.SetMin(300);
```

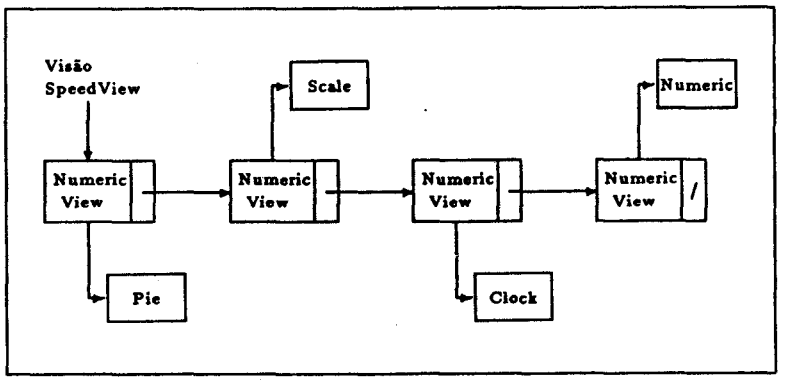

Figura 5.3: Estrutura de dados necessária para representar o nível de controle sobre os objetos interativos

Mensagens como estas não seriam recomendáveis, pois o controle nelas embutido está tratando diretamente com os objetos envolvidos de uma aplicação particular. A adoção desta situação implicaria na mudança do código do **controle** sempre que houvesse a criação de um novo objeto interativo, ou pior: sempre que se resolvesse substituir os agentes da interação.

Este "problema" poderia ser solucionado eliminando-se do **controle** a "visão" sobre os objetos interativos envolvidos. Em programação orientada a objetos e, em particular em C++, esta solução é resolvida através da adoção do **polimorfismo**. Em C++ quando se desejar que um método tenha comportamento polimórfico **em relação às suas subclasses**, acrescenta-se, antes da especificação do mesmo, a palavra reservada `virtual`[6]. Por este motivo, todos os métodos definidos na classe `NumericView` da figura 5.1 foram especificados como **virtuais**, tal como ilustrado no apêndice A. A implementação do **controle** fica agora mais simples: ao invés de serem instanciados objetos interativos específicos (`Pie`, `Scale`, `HorizontalMeter`, ...) no seu código, apenas referências a **objetos genéricos** do "tipo" `NumericView` é que seriam criados e, durante a execução do módulo, as mensagens polimórficas seriam "enviadas" diretamente aos objetos referenciados por estes objetos genéricos (`NumericView`).

Por outro lado, foi necessário implementar uma estrutura de dados auxiliar para representação dos objetos interativos. Uma estrutura simples, como a mostrada na figura 5.3, será suficiente para esta representação.

A figura 5.3 ilustra uma lista encadeada onde cada nó da lista contém um apontador para a instância do objeto interativo definido para aquela visão e um apontador para o próximo elemento da lista.

[6]ver subseções 2.4.7 e 2.4.8.

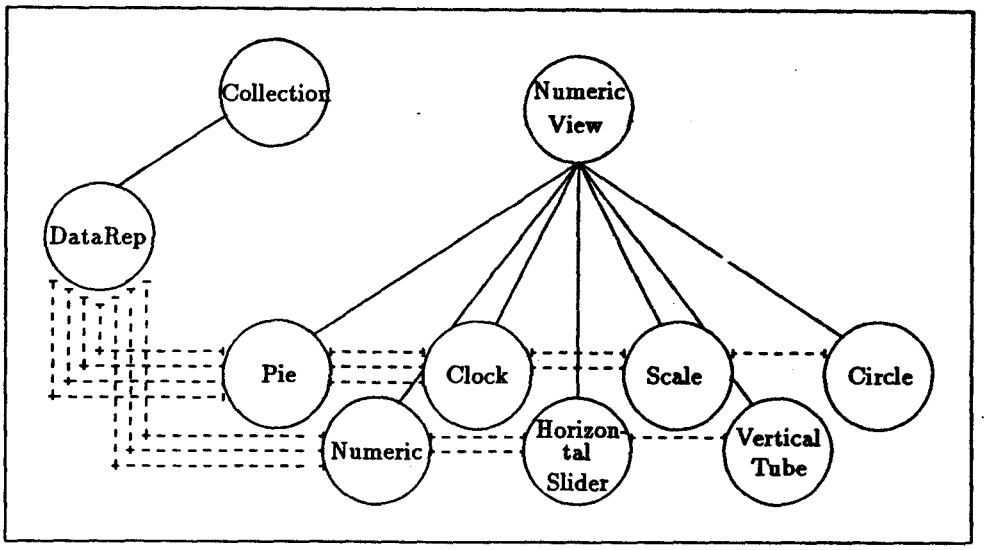

Figura 5.4: Nova hierarquia de abstrações do princípio reativo

A lista apresentada nesta figura representa os objetos interativos definidos para uma visão de objetos chamada `SpeedView`. Esta visão possui quatro objetos em sua representação (`Pie`, `Scale`, `Clock` e `Numeric`) mas, como observado, apenas um "tipo" de objeto faz parte da lista genérica: `NumericView`. O conteúdo desta lista pode, inclusive, mudar durante a execução da aplicação. Ou seja, pode ser inserido mais um objeto interativo, ou mesmo substituído, ou removida alguma representação.

A classe que implementa o **controle** do princípio reativo, que nesta implementação é chamada de `DataRepresentation` (ou simplesmente `DataRep`), é inserida na hierarquia da figura 5.1 como uma subclasse de uma classe que implementa operações padrões sobre listas encadeadas (a classe `Collection`). A nova hierarquia é mostrada como na figura 5.4. Os segmentos de reta tracejados simbolizam as operações abstratas de **agregação** e **refinamento**.

A especificação das classes `DataRep` e `Colletion` estão ilustradas no apêndice A.

A nova hierarquia de classes definida acrescenta à já existente a definição do nível de **controle** (o comportamento da classe `DataRep`) e a definição da chamada **abstração central** da figura 4.10

Para ilustração do funcionamento da classe `DataRep`, a figura 5.5 possui um fragmento de código dos métodos `InsertNewView` e `ChangeVal`, possibilitando respectivamente a adição de um novo objeto interativo aos já existentes, e o envio de mensagens em *"broadcast"* para todos os objetos relacionados àquele controle, autorizando um reajustamento de valores.

```
void DataRep :: InsertNewView(NumericView *MyNumView)
{
   Put(MyNumView);
   UpdateNextPos(MyNumView);
   MyNumView->Draw(ViewWindow,GcOutline,GcFill);
}
void DataRep :: ChangeVal(int Value)
{
   NumericView *MyNumView = Start();

   while (MyNumView != NULL) {
      MyNumView->ShowValue(ViewWindow, GcOutline, GcFill, Val);
      MyNumView = Succ();
   }
}
```

Figura 5.5: Parte da implementação da classe `DataRepresentation`

É interessante observar, nesta figura a inexistência no código de qualquer objeto interativo específico. Em **todos os métodos** do código apenas os objetos genéricos `NumericView` são mencionados. A definição **de quais os objetos interativos específicos** irão participar da interação partirá exclusivamente da aplicação propriamente dita.

O comportamento de alguns dos outros métodos relativos ao **controle** do princípio reativo (*i.e.* rejeição de valores, mudança do escopo de validade da representação, mudança do valor corrente partindo da aplicação dentre outros) será mostrado nas subseções seguintes.

5.1.3 Rejeição de valores

O funcionamento da hierarquia de abstrações é bastante similar ao funcionamento do modelo PAC, descrito nas seções anteriores. Algumas mudanças "superficiais" foram introduzidas para que o modelo fosse adaptado às condições impostas pelo C++.

A rejeição de valores, como visto, pode acontecer em duas situações:

- rejeição pela aplicação, significando que o novo valor solicitado pelo usuário não é permitido para aquele contexto de execução e,
- rejeição por parte do modelo PAC, significando que o novo valor solicitado está fora do limite de validade permitido para aquele objeto.

Apenas o segundo caso "interessa" neste momento da discussão. A descrição da rejeição de valores para um determinado contexto de execução será realizada na elaboração da aplicação propriamente dita.

```
void DataRep ::  RequestNewData(int MouseRow,int MouseCol)
{
   ...
   while (MyNumView != NULL) {
      if (MyNumView->InsideBox(MouseRow,MouseCol) == TRUE) {
         Value = MyNumView->Get(MouseRow,MouseCol);
         break;
      }
      MyNumView = Succ();
   }
   if (Value < Min || Value > Max)
      //Reject Change
   ...
}
```

Figura 5.6: Rejeição de valores na classe **DataRepresentation**

A figura 5.6 ilustra um fragmento de código em C++ que implementa o controle da rejeição de valores pelo modelo PAC. O método ilustrado faz parte da classe **DataRep** e a parte sublinhada corresponde, no modelo PAC, à consulta que o controle central faz à abstração central sobre o domínio de validade do objeto.

Durante a requisição de um novo dado, se o novo valor que está solicitado (**Value** na figura 5.6) estiver fora dos limites de validade daquela representação de dados, então ele deve ser desprezado.

5.1.4 Mudança do escopo de validade

A mudança do escopo de validade (limites mínimo e máximo) de uma representação de dados ocorre **sempre** por parte da aplicação. Ou seja, como descrito no modelo PAC da figura 4.10, esta requisição de mudança parte da **aplicação** em direção ao **controle central**. O controle central, por sua vez, precisa mudar o limite na **abstração central** e nas **subabstrações** que possuem, em sua memória, informações sobre este limite. A mudança nas **subabstrações** é feita pelos **subcontroles** correspondentes.

A implementação desta propriedade foi dividida, para efeito didático, em duas partes similares:

- mudança do limite **mínimo** e
- mudança do limite **máximo**.

Será considerado, para ilustração da propriedade, a mudança do limite **máximo**. A figura 5.7 ilustra a implementação da mudança do limite máximo pelo **controle central**

```
void DataRep ::  ChangeMax(int Maximum)
{
   Max = Maximum;
   NumericView *MyNumView = Start();
   while (MyNumView != NULL) {
      MyNumView->SetMax(ViewWindow,GcOutline,GcFill,Maximum);
      MyNumView = Succ();
   }
}
```

Figura 5.7: Mudança do limite máximo de uma variável do **controle** para a **abstração central**

```
void HorizontalMeter ::  SetMax(...,int Maximum)
{
   Max = Maximum;
   ShowValue(ViewWindow,GcOutline,GcFill,Maximum);
}
```

Figura 5.8: Mudança do limite máximo da variável do subcontrole para as subabstrações

na **abstração central**. Ou seja, o comportamento da classe **DataRep** em relação à alteração de sua memória. É interessante **enfatizar** que este método é executado a partir da solicitação da **aplicação**.

Uma propriedade interessante desta figura é que a mudança do limite máximo do objeto (ou desta representação de dados) ocorre na primeira linha do código (parte sublinhada), e o restante do código ilustra apenas as mensagens em *"broadcast"* ou **polimórficas** que o controle central envia para os subcontroles, "ordenando" a mudança do limite máximo nas subabstrações correspondentes.

A figura 5.8 ilustra a recepção desta "ordem", por parte do controle central, sobre a mudança no limite máximo da instância do **objeto interativo** específico.

O fragmento de código ilustrado na figura faz parte da classe que implementa o objeto interativo **HorizontalMeter**. Um fato ilustrado com relevância notada é que, além da mudança na subabstração por parte deste subcontrole, ocorre também um reajustamento na visão externa associada. A razão disto é intuitiva, uma vez que, modificado o escopo

```
int DataRep ::  RequestNewData(int MouseRow,int MouseCol)
{
   ...
   Value = MyNumView->Get(MouseRow,MouseCol);
   ...
   return Value;
}
```

Figura 5.9: Requisição de um novo dado por parte da aplicação

de validade do objeto, ele **precisa** ter o seu **valor** rearrumado para as novas dimensões. Isto é feito com o método `ShowValue`.

5.1.5 Propagação de valores

A propagação de novos valores sendo solicitados deve acontecer em "direção" à aplicação em uma primeira etapa, e caso este valor seja permitido, para o contexto de execução sendo modificado: a aplicação retransfere a gerência da atualização para o controle central, que então autoriza a mudança em *"broadcast"* para todas as visões.

A figura 5.9 ilustra a transferência de controle da classe `DataRep` para a aplicação e a figura 5.10 a atualização de todas as visões externas dos objetos interativos envolvidos.

No funcionamento, a aplicação simplesmente solicita um novo valor (método `RequestNewData`), e o controle verifica em qual visão o usuário está interagindo com a aplicação. De posse do novo valor obtido da visão, o controle transfere a execução para a aplicação (`return Val`), que por sua vez verifica a consistência desta informação e, caso seja permissível, retransfere o domínio para o controle, que então atualiza as visões.

5.2 Independência do princípio reativo

Um dos principais atrativos na utilização do modelo PAC e da sua implementação é a possibilidade de se produzirem aplicações localizadas num patamar diferente dos níveis de representação do princípio reativo.

Para a criação de uma representação de um objeto com visões, a quantidade de código produzido pela aplicação é mínima. Para ilustração do código **efetivamente** produzido pela aplicação será considerado o exemplo da figura 5.11, que foi produzido pela implementação disponível. Será feita a simulação ou interação das representações de um objeto numérico qualquer (que, neste exemplo, foi chamada de `SpeedView`). As quatro visões que a aplicação deseja interagir são `Pie`, `Numeric`, `VerticalTube` e `HorizontalSlider`,

```
int DataRep ::  UpdateChange(void)
{
   NumericView *MyNumView = Start();
   while (MyNumView != NULL) {
      MyNumView->ShowValue(...,Val);
      MyNumView = Succ();
   }
}
```

Figura 5.10: Propagação de valores do controle para as visões

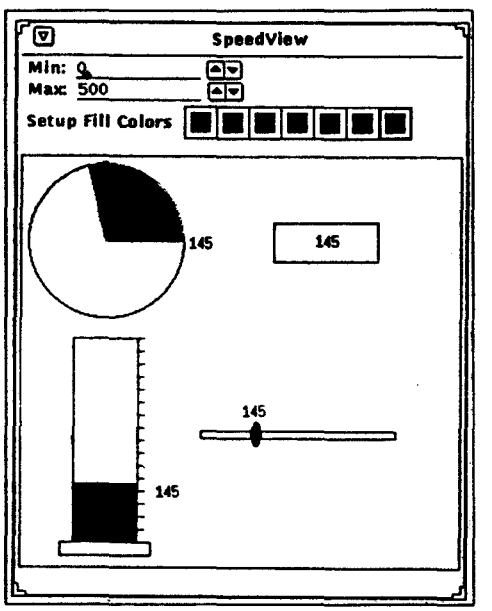

Figura 5.11: Representação do Objeto SpeedView em suas quatro visões externas

```
DataRep *SpeedView = new DataRep("SpeedView",Min,Max,Min);
...
TmpView = new Pie(Min,Max,60,AUTOMATIC);
SpeedView->InsertNewView(TmpView);
TmpView = new Numeric(DEFINEDBYUSER,50,195);
SpeedView->InsertNewView(TmpView);
TmpView = new VerticalTube(Min,Max,160,AUTOMATIC);
SpeedView->InsertNewView(TmpView);
TmpView = new HorizontalSlider(Min,Max,150,DEFINEDBYUSER,190,135);
SpeedView->InsertNewView(TmpView);
```

Figura 5.12: Código produzido pela aplicação para gerar o princípio reativo

tal como mostrado na figura. O fragmento de código da figura 5.12 corresponde exatamente ao código necessário para produzir o **controle**, a **abstração** e a **representação** do princípio reativo.

Na análise deste código, vemos que, para utilização do princípio reativo, basta apenas que uma representação de visões ou dados seja criada e que se especifique quais os objetos interativos que devem fazer parte da interação. O restante do trabalho é realizado pelas classes que implementam o princípio reativo e pelo *event handler* do XWindows (ou do XView). Ou seja, o gerenciador de *loop* de eventos do XWindows "detecta" um *click* ou um *drag* de *mouse* e, caso se esteja interagindo com as visões de objetos, a execução é transferida para o controle do princípio reativo.

A criação de um objeto que terá o comportamento do controle do princípio reativo é feita através do comando: `SpeedView = new DataRep("SpeedView",Min,Max,Min)`. O primeiro parâmetro utilizado na criação do objeto refere-se ao nome da visão exposta no *frame* e os outros três parâmetros dizem respeito à criação deste objeto, à abstração central das visões: os limites mínimo e máximo assumidos pelo objeto a ser interagido e o valor inicial a ser exibido.

Os outros **únicos pares** (quatro no total) de comandos dizem respeito à definição de quais visões serão utilizadas na interação.

O primeiro par cria a visão da **torta gráfica** com limites de validade `Min` e `Max`, com raio de valor 60 *pixels* e com alocação de posição automática. Depois de criada a visão, o passo seguinte é a **inserção** desta visão na lista de visões ativas da representação de dados sendo especificada. Isto é feito através do comando `SpeedView->InsertNewView(TmpView)`. Os demais pares de comandos criam as outras três visões e possuem comportamento similar. É interessante constatar que no par de comando referente à criação da visão do

mostrador digital (`Numeric`) não é passado como parâmetro o escopo de validade do objeto a ser modificado, e também que a posição da janela onde estará localizada esta visão foi **definida pelo usuário** através do atributo `DEFINEDBYUSER`, seguido das coordenadas cartesianas desejadas. A utilização deste atributo faz com que o usuário, projetando a aplicação, tenha uma "preocupação a menos" na definição do espaço.de interação, uma vez que, se for especificado que a alocação de posição dentro da janela de interação será automática (tal como a criação de *panel items* em *panels* do XView - o usuário vai criando os botões, *menus*, etc. e caso ele não diga explicitamente a posição onde queira inserir os itens, o sistema é que se encarrega da alocação física dentro do *panel* definido), o controle do princípio reativo é que se encarrega da alocação.

5.3 Uma extensão no modelo de objetos para permitir reatividade composta

5.3.1 Introdução

A implementação do princípio reativo, mostrada nas seções anteriores, trata apenas da atualização de múltiplas visões referentes a uma **mesma entidade**, ou uma **única representação de dados**. Uma extensão, que, no paradigma de objetos possui implementação trivial, é a que trata da atualização de visões de objetos representando diferentes entidades de memória. A implementação desta propriedade irá permitir que sistemas reativos, como os ilustrados nas figuras 2.22 e 4.2, possam ser implementados de uma forma tão trivial quanto os sistemas reativos produzidos pelo esquema da seção anterior. Uma primeira "diferença", que claramente estes "novos" terão em relação aos até aqui definidos, é a de que eles terão múltiplas representações de dados (ou seja, vários objetos do "tipo" `DataRep`). Uma outra propriedade que estes novos sistemas terão é a necessidade de criação de uma estrutura que possa associar as representações criadas. Vale dizer: a criação de uma espécie de **relação** entre as representações, composta das entidades envolvidas e da definição da relação propriamente dita.

5.3.2 Um exemplo didático

A implementação do princípio reativo estendido para suportar múltiplas visões, será mostrada através de uma aplicação didática e a sua implementação em C++. As simplificações e imperfeições deste exemplo são inevitáveis, por se tratar de uma aplicação puramente didática.

O problema a ser implementado é a simulação de um **movimento uniformemente variado (MUV)**, cujas equações de representação são:

1. $s = s_0 + v_0 t + \frac{1}{2} a t^2$
2. $v = v_0 + at$

Uma primeira constatação mostra que teremos seis variáveis controladas pela aplicação e, conseqüentemente, seis visões associadas (seis objetos do tipo `DataRep`):

- velocidade inicial
- distância inicial
- aceleração
- tempo
- velocidade final
- distância percorrida

Existe ainda uma relação de dependência entre a representação da **velocidade** e as representaçõs da velocidade inicial, da aceleração e do tempo; e uma outra dependência entre a representação da **distância percorrida** e as representações da distância inicial, da velocidade inicial, da aceleração e do tempo. As outras quatro representações (aceleração, tempo, distância inicial e velociadade inicial) não possuem relação com nenhuma outra[7], e por isso mesmo têm comportamento "livre".

Uma conclusão imediata da identificação das relações de dependência entre as representações é a de que a mudança de valores nas visões de velocidade inicial ou nas visões de aceleração, ou ainda nas visões de tempo, provocará uma atualização nas visões da velocidade final. O mesmo ocorre quando há uma atualização nas visões de uma das representações dependentes da representação da distância percorrida.

Várias formas de representação destas relações foram idealizadas. Uma preocupação inicial na escolha desta representação era a de que a relação pudesse ser facilmente modificada pelo usuário. Uma solução que possui algumas imperfeições, mas que é extremamente flexível, é a que deixa a cargo do usuário a criação e implementação de uma função, cujo código possui a relação de dependência entre as visões pretendidas, e cujo endereço é passado para o princípio reativo. Para o exemplo da simulação do MUV, a implementação de uma das funções poderia ser como a mostrada na figura 5.13. O esquema da adaptação ou do "acoplamento" das funções de relação definidas pelo usuário ao princípio reativo será mostrada na seção seguinte.

[7] em outras palavras, não dependem de outras representações para terem seus valores atualizados.

```
int DistanceFunction(void)
{
   int s0 = InitialDistanceView->GetVal();
   int v0 = InitialSpeedView->GetVal();
   int a = AccelerationView->GetVal();
   int t = TimeView->GetVal();
   int s = s0 + v0*t a*t*t/2;

   return s;
}
```

Figura 5.13: Implementação de uma das funções de relações do MUV

5.3.3 A classe Relation

Esta classe, como o próprio nome já sugere, associa as funções definidas pelo usuário ao princípio reativo. A hierarquia de abstrações definida na figura 5.4[8] "sugere" que o controle do princípio reativo é a classe **DataRep**, e que a instanciação da mesma na aplicação é suficiente para implementação do princípio reativo. Entretanto, a atualização de múltiplas visões de objetos distintos não era implementada.

A classe **Relation** simplesmente associa uma representação de dados (**DataRep**) com um endereço de função, onde está implementada a sua relação de dependência com as outras representações. Então, o conteúdo de um objeto instanciado da classe **Relation** nada mais é do que um conjunto ou uma **coleção** de objetos de representação de dados com seus respectivos endereços de funções que implementam as relações de dependência. Para o exemplo de simulação do MUV, descrito na seção anterior, o conteúdo do objeto derivado da classe **Relation** poderia ser como o mostrado na figura 5.14.

A classe **Relation** (assim como a classe **DataRep**) utiliza as operações abstratas de manipulação sobre listas encadeadas (inserção, modificação, etc.) e por este motivo ela também será subclasse da classe **Collection**. E, na sua memória principal, há **agregações** de objetos do tipo **DataRep**. Desta forma, a hierarquia de abstrações do princípio reativo da figura 5.4 foi redefinida e está ilustrada na figura 5.15. A aplicação está representada por um retângulo, e a linha tracejada, como no modelo anterior, representa operações de **agregação** e **refinamento**, e a linha pontilhada representa a **instanciação** de uma **relação** na aplicação.

A figura 5.16 ilustra a implementação da simulação do MUV com as classes desenvolvidas para este trabalho. O esquema de atualização de valores neste sistema funciona de

[8] ver subseção 5.1.2.

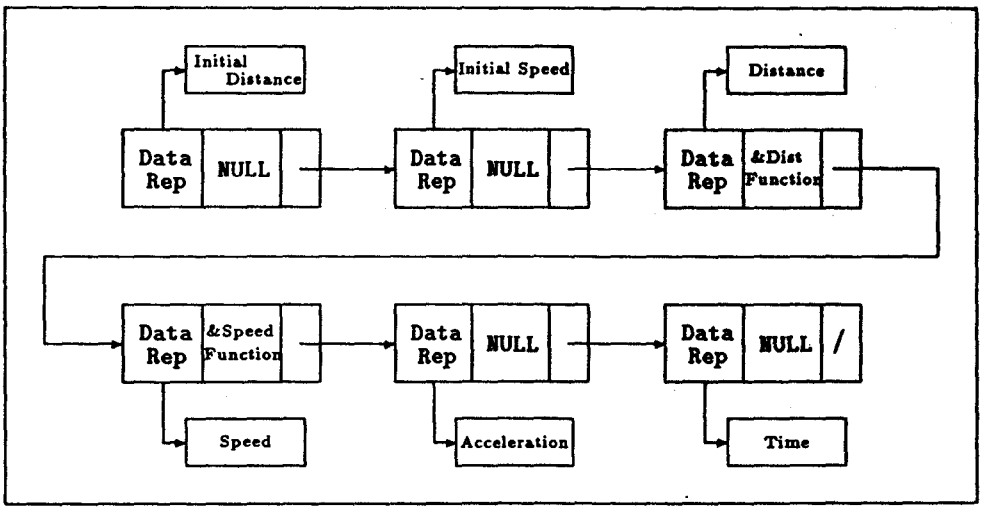

Figura 5.14: Estrutura de dados do objeto `MUV_Relation`

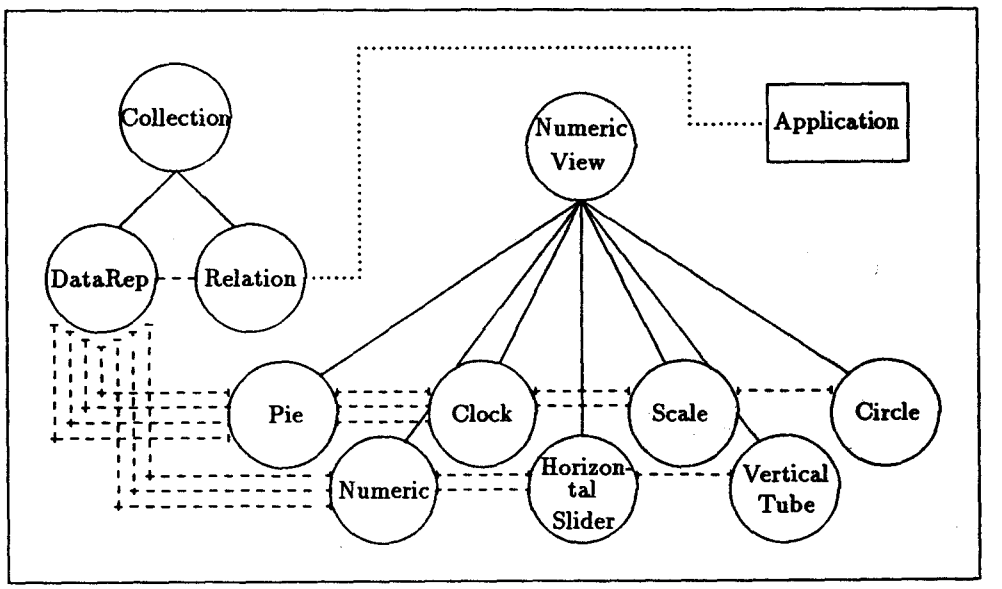

Figura 5.15: Hierarquia de abstrações do princípio reativo estendido

forma análoga (agora estendida) aos sistemas descritos neste capítulo.

5.3.4 Problemas de circularidade e redundância

Uma das falhas da implementação da classe `Relation` está relacionada diretamente com a flexibilidade permitida ao usuário na criação das funções de relação entre as representações de dados e na criação do objeto do tipo `Relation` propriamente dito.

No exemplo da simulação do MUV, apenas duas relações foram criadas. As demais estavam "livres" e foram definidas, na criação do objeto do tipo `Relation`, como **nulas** ou **vazias** (`NULL`). Se, por uma falha do usuário, ele definisse alguma relação para, pelo menos, uma das quatro representações **nula**, o sistema entraria numa **circularidade**, e informações **redundantes** seriam produzidas. Por exemplo: foi definido que a equação da velocidade era $v = v_0 + at$. Se, desta equação, o usuário produzir uma nova relação (tal como: $v_0 = v - at$), o sistema iria calcular resultados redundantes[9] pois faria com que a velociadade inicial se tornasse função da velocidade final, da aceleração e do tempo. O que entra em conflito com a definição de que a velocidade inicial em um MUV é uma constante, e não possui dependência com nenhuma outra. De qualquer forma, vale salientar que este problema foi criado por uma "falha" do usuário na especificação das relações, problema decorrente exclusivamente da intenção em tornar o sistema o mais flexível possível para o mesmo.

[9] inesperados?

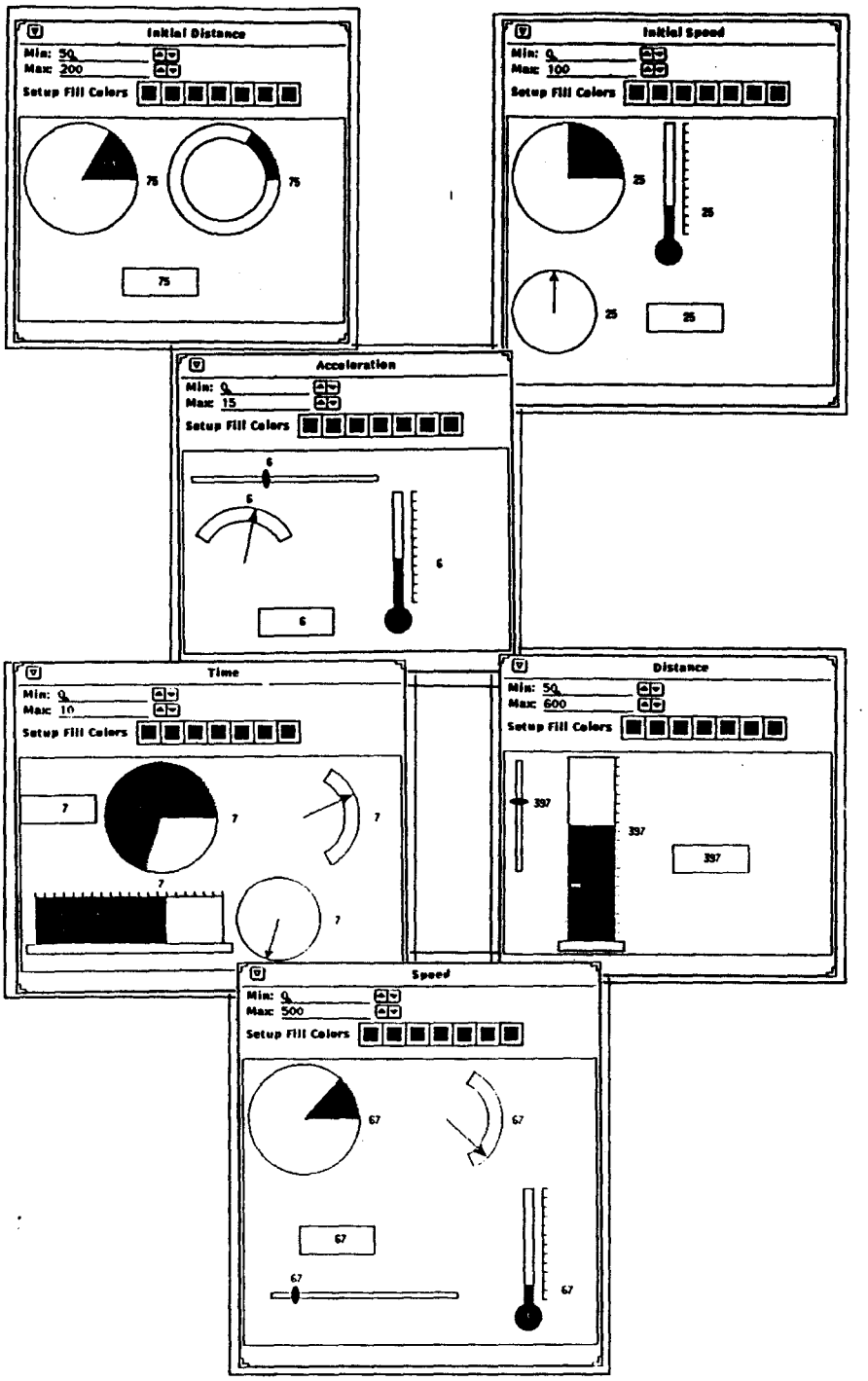

Figura 5.16: Aspecto das representações da simulação do MUV

Capítulo 6

Conclusões

6.1 Utilização de toolkits no desenvolvimento de interfaces

O desenvolvimento de software (e de interfaces) passou a ser, nos últimos anos, uma das atividades mais dispendiosas, em qualquer ramo da automação de tarefas. Cada vez mais a produção de programas vem se tornando mais onerosa, em termos de custos de mão de obra, tempo de desenvolvimento e de manutenções, e a nível de sofisticação dos produtos desenvolvidos. Ou seja, o "grande desafio" do final da década de 1980[1], citado no capítulo 2, surgiu como "tentativa" de diminuir a distância que havia entre a realidade dispendiosa, na concepção e implementação de programas, e a "popularidade" (e rapidez) pretendida para o desenvolvimento dos mesmos.

O primeiro passo na direção da tentativa de redução desta distância veio com a utilização dos chamados sistemas *toolbox* no desenvolvimento de programas. O uso de sistemas *toolbox* tornou rápido e barato o desenvolvimento e a prototipagem dos programas. No entanto, a flexibilidade, ou seja, a possibilidade de o projetista poder fazer ligeiras modificações nos sistemas, para melhor adequá-los às aplicações, ainda não havia sido atingida. Cada vez mais os programas produzidos não seguiam uma consistência em relação aos já existentes, e os "blocos pré-fabricados" de componentes de construção eram rígidos e de difícil modificação. Quase sempre o projetista se via às voltas com duplicação de código, programas cada vez maiores e, conseqüêntemente, de difícil manutenção.

A chegada dos *toolkits* simplificou este problema, de vez que a adoção desta técnica quase sempre induzia a utilização dos conceitos de abstração e representação de dados na concepção de programas em desenvolvimento e também na utilização do estilo de programação orientado a objetos. Os conceitos de herança, polimorfismo, ocultamento de informações e abstração de dados levaram o desenvolvimento de interfaces a patamares de simplificação e representação bem diferentes dos encontrados na filosofia *toolbox*. No projeto da interface tornou-se possível, agora, fazer modificações em um *menu* ou em

[1] construção de software rápida, barata e flexível (ver subseção 2.1.1).

uma janela de interação para melhor adequá-los a um contexto de comunicação entre o homem e a máquina. A **reutilização de código**, descrita no capítulo 2, trouxe a simplificação e diminuição do tamanho do código fonte das aplicações. O uso da propriedade de polimorfismo[2] entre objetos trouxe a escrita de código abstrato nos seus métodos. Por fim, a herança de propriedades (e de classes) trouxe a versatilidade e a melhor modelagem das características dos componentes interativos no projeto das interfaces. A resposta a tudo isto foi, inevitavelmente, a construção e o desenvolvimento de interfaces (e de programas num sentido geral) de uma maneira rápida, barata e flexível. Acredita-se que a adoção de *toolkits* no desenvolvimento dos programas levou à diminuição, em até 80% [Wei89], do tamanho e do tempo de concepção dos programas fonte, em comparação ao desenvolvimento com as técnicas convencionais.

Uma ilustração desta redução de tamanho de código fonte destes softwares é mostrado na implementação do exemplo da figura 5.11 do capítulo 4. A criação dos objetos, incluindo os níveis de controle, apresentação e abstração do princípio reativo, **no programa definido pelo usuário**, possui apenas **nove** linhas de código fonte, as quais correspondem à criação dos objetos interativos necessários à utilização do sistema reativo sendo criado. O único "trabalho" do programador é o de decidir quais as peças de interação que serão utilizadas na montagem de um sistema reativo. Além da redução significante do tamanho e do tempo de desenvolvimento que os sistemas *toolkits* traziam, uma outra propriedade na sua utilização foi a nítida separação entre os aspectos específicos da aplicação (rotinas e procedimentos, lidando exclusivamente com aspectos da **funcionabilidade** da aplicação), e os aspectos de interação com o usuário (no caso particular, com o princípio reativo sendo construído).

6.2 Opções e problemas surgidos

O ET++ (Editor Toolkit) é constituído por uma biblioteca de classes orientada a objetos, a qual implementa os principais componentes de blocos de construção de interfaces homem-máquina. O ET++ foi desenvolvido na Universidade de Zürich e, atualmente, é formado por uma hierarquia que comporta aproximadamente 140 classes escritas em C++. O principal objetivo dos idealizadores do ET++, na construção desta hierarquia, foi a possibilidade de concepção de aplicações, de uma maneira rápida, homogênea e consistente, seguindo a metáfora *desktop* de desenvolvimento de interfaces. Assim como o X11, o ET++ foi desenvolvido sob uma interface abstrata de comunicação com o gerenciador de janelas. Desta forma, o sistema se comunica apenas com este nível abstrato, enquanto que a comunicação com o sistema de janelas específico, bem como a comunicação com o sistema operacional, fica a cargo da interface abstrata.

Todos os pré-requisitos para adoção do ET++ como o *toolkit* padrão, sob o qual o princípio reativo seria estendido, estavam satisfeitos: o sistema de janelas disponível era

[2]ver subseções 2.4.7 e 2.4.8.

o X11R4; os fontes e as bibliotecas estavam disponíveis, através de FTP (File Tranfer Protocol)[3]; a linguagem de interface era o C++; o sistema havia sido desenvolvido numa Sun Sparc Station "rodando" o sistema Unix Sun OS release 4.0; a hierarquia de abstrações definida implementava algo em torno de 140 componentes e, por fim, o software era de domínio público.

Feita a "aquisição" do produto, modificados os *makefiles* e a instalação no equipamento, o próximo passo seria a impressão dos manuais. A inexistência, na época, de uma impressora laser padrão PostScript no departamento, e alguns problemas encontrados na descompactação do manuais, fez com que fosse pedido um *hardcopy* da documentação na Suíça[4]. Devido provavelmente à recente publicação do produto[5], a documentação disponível era bastante precária. A maioria dos textos remetidos e disponíveis em publicações internacionais constituia-se de resumos das apresentações em palestras, seminários, etc. Ainda não tinham sido preparados os manuais de referência do sistema e uma especificação mais detalhada dos componentes.

Algumas tentativas, sem sucesso, foram feitas no sentido de depurar os 8 megabytes de código fonte (aproximadamente 200.000 linhas de implementação), disponíveis na versão 2.2. Infelizmente, a documentação ainda é um requisito primordial na adoção de um produto servindo de base para o desenvolvimento de um projeto computacional de qualquer natureza.

6.3 Opção pelo XView Toolkit

O XView é um *toolkit* utilizado principalmente para desenvolvimento de interfaces homem-máquina o qual foi desenvolvido pela Sun Microsystems e pela AT&T para ser uma espécie de interface padrão do Unix System V Release 4[6]. Apesar do XView ter sido implementado sob um sistema de janelas padrão (XWindows), e uma arquitetura específica (Sun Workstations), foi ele o sistema base escolhido para a implementação do problema proposto, uma vez que já havia uma versão instalada no Departamento de Ciência da Computação da Unicamp e, além disto, existia também uma certa "tendência" para a utilização do *toolkit* em outros projetos em desenvolvimento; e, o mais importante, havia disponível uma "quantidade de páginas" razoável de documentação sobre a sua utilização. A opção pelo *toolkit* poderia ter sido em direção a outros sistemas na linha do ET++, como por exemplo o InterViews [LCV87]. Entretanto, o receio e a "experiência negativa" sofrida com

[3]gostaria de agradecer ao prof. Paulo Lício de Geus (DCC-IMECC-UNICAMP) pela documentação fornecida sobre o *ftp server* disponível na Universidade de Princeton, Inglaterra, sem a qual a aquisição do produto teria se tornado "impossível".

[4]gostaria de agradecer a Erich Gamma (IFI-UZ), um dos três criadores do ET++, por enviar as últimas versões da documentação do *toolkit*.

[5]a versão 2.2 do ET++ foi publicada em 21 de outubro de 1990, na European Conference on Object-Oriented Programming (ECOOP-OOPSLA), em Ottawa, Canadá.

[6]ver seção 3.1.

```
Panel_item PanelText;

PanelText =
      (Panel_item)xv_create(panel, PANEL_TEXT,
      PANEL_LABEL_STRING,          "Gate Values:",
      PANEL_VALUE,                 "",
      PANEL_VALUE_DISPLAY_LENGTH,  30,
      PANEL_VALUE_STORED_LENGTH,   500,
      PANEL_NOTIFY_PROC,           GetValuesProcedure,
      NULL);
```

Figura 6.1: Especificação do *panel item* para objetos do "tipo" `PreDefined Gate`

o ET++, fez com que a opção fosse feita em direção a algo mais concreto.

6.4 O esquema das callback functions

Um dos problemas inicialmente encontrados no esquema que o XView introduzia era o relacionado com a implementação das *callback functions*. Quando o identificador de uma *callback function* é definido na especificação da interface[7], isto vale dizer que o XView irá inicializar uma tabela interna e, também, que irá informar ao *notifier* que aquela função deve ser executada quando da interação do usuário com o componente sendo especificado. O problema é que, na implementação e posterior execução do código da *callback function*, o projetista não tem mais controle sobre qual o objeto que chamou aquela função, e conseqüêntemente, se esvai o esquema de modularidade de classes, introduzido pela programação orientada a objetos. Um exemplo deste problema pode ser ilustrado na figura 6.1 onde é mostrada a especificação de um "objeto" do tipo `PANEL_TEXT`. O problema resume-se na seguinte descrição: O simulador de circuitos desenvolvido possui um componente chamado `PreDefined`, que é um tipo especial de injetor o qual, como o próprio nome já sugere, possui valores pré-determinados e definidos pelo usuário, por ocasião de sua criação. Uma solução para a implementação deste fato seria a especificação do *panel item*, ilustrado na figura 6.1, em conjunto com a definição de todos os outros componentes da interface (no programa principal). O problema encontrado, neste caso, foi que na criação de objetos instanciados da classe `PreDefined` e, conseqüêntemente, na especificação dos pulsos elétricos pré-determinados pelo usuário, o *notifier* desvia o controle do programa para a *callback function* definida (em particular a função `GetValuesProcedure`) pelo usuário. A partir daí, o usuário não possui mais controle sobre qual objeto foi responsável pela

[7] ver seção 3.2.

```
Class PreDefined {
   ...
public:
   ...
   void GetValuesProcedure(Panel_item,Event *)
   ...
};
PreDefined ::  PreDefined(int X,int Y,...)
{
   PanelText =
        (Panel_item)xv_create(panel, PANEL_TEXT,
              ...
              PANEL_VALUE_STORED_LENGTH, 500,
              PANEL_NOTIFY_PROC,         GetValuesProcedure,
              NULL);
              ...
}
```

Figura 6.2: Adaptação das *callback functions* na especificação e na definição da classe `Predefined`

chamada da função, uma vez que a interface entre a *callback function* e a especificação do objeto é fixa, ou seja, não é possível especificar o número de parâmetros, e tampouco quais os parâmetros que poderiam ser passados para a função. A conseqüência imediata deste fato foi a criação de procedimentos extras a serem produzidos, a fim de que, na *callback function*, em questão, pudesse ser "decoberto" qual o objeto que a chamou.

Solução menos penosa para este problema seria a adaptação de um *callback function* como um **método** definido no comportamento de uma classe. Neste caso, o *notifier* só teria acesso ao identificador da *callback function* por ocasião da criação de um objeto `PreDefined`. Esta solução, aparentemente ideal, não é possível de se implementar, pois o identificador da *callback function*, no esquema criado pelo XView, precisa ser conhecido por ocasião da compilação dos módulos. Ou seja, o XView tem que "conhecer" o endereço da *callback function* associada ao objeto em definição em tempo de compilação o que, infelizmente, faz com que o esquema introduzido na utilização do paradigma de objetos não possa ser incorporado integralmente. Em resumo: **a utilização do XView em aplicações desenvolvidas sob o C++ não é feita de forma natural.**

Apêndice A

Modelo e Interface das Classes do Princípio Reativo

A.1 Classe NumericView

A.1.1 Especificação

```
// NumericView.h

#ifndef NUMERICVIEW
#define NUMERICVIEW

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include "Definition.h"

typedef struct _RectBox {
    int X1;
    int Y1;
    int X2;
    int Y2;
} RectBox;

class NumericView {
protected :
    int Row;
    int Col;
    int Val;
```

```
    int StorageMode;
public :
    NumericView(int SM,int R=0,int C=0,int Value = 0);
    virtual void    SetCoords(Point &);
    virtual void    GetCoords(Point &);
    virtual void    GetSize(Point &) = 0;
    virtual void    Draw(Window,GC,GC) = 0;
    virtual void    SetMin(Window,GC,GC,int) { }
    virtual void    SetMax(Window,GC,GC,int) { }
    virtual void    ShowValue(Window,GC,GC,int) = 0;
    virtual int     Get(int,int) = 0;
    virtual boolean InsideBox(int,int) = 0;
    virtual int     GetStoMode(void) { return StorageMode; }
    void            SetValue(int Value) { Val = Value; }
};

#endif
```

A.2 Classe Circle

A.2.1 Visão Externa

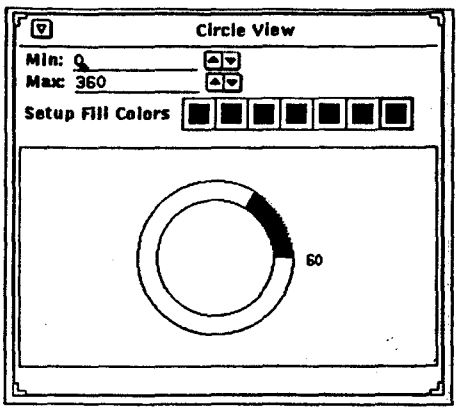

Figura A.1: A classe **Circle**

A.2.2 Especificação

```
// Circle.h

#ifndef CIRCLE
#define CIRCLE

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "Definition.h"
#include "NumericView.h"

class Circle : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     RadiusSmall;
    int     RadiusBig;
    RectBox CircleBox;
    GC      GcWhite;
public :
    Circle(int Minimum=0,int Maximum=360,int RadSmall=30,
           int RadBig=45,int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetCenter(Window,GC,GC,int,int);
    void    SetRadiusSmall(Window,GC,GC,int);
    void    SetRadiusBig(Window,GC,GC,int);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.3 Classe Clock

A.3.1 Visão Externa

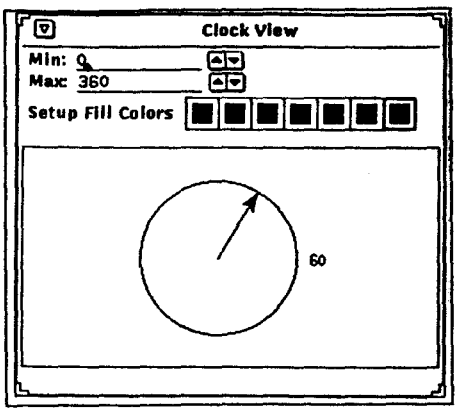

Figura A.2: A classe **Clock**

A.3.2 Especificação

```
// Clock.h

#ifndef CLOCK
#define CLOCK

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "Definition.h"
#include "NumericView.h"

class Clock : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     Radius;
    RectBox ClockBox;
public :
    Clock(int Minimum=0,int Maximum=360,int Rad=45,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetCenter(Window,GC,GC,int,int);
    void    SetRadius(Window,GC,GC,int);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.4 Classe Horizontal Meter

A.4.1 Visão Externa

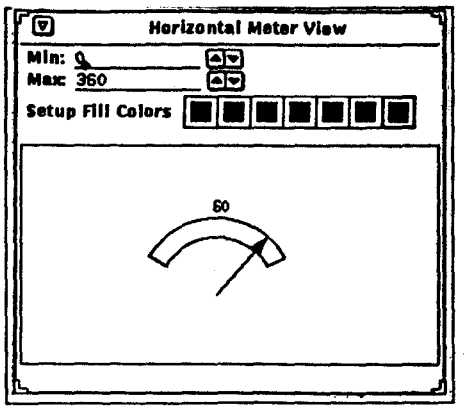

Figura A.3: A classe **Horizontal Meter**

A.4.2 Especificação

```
// HorMeter.h

#ifndef HORMETER
#define HORMETER

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "Definition.h"
#include "NumericView.h"

class HorizontalMeter : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     RadiusSmall;
    int     RadiusBig;
    RectBox HorMeterBox;
public :
    HorizontalMeter(int Minimum=0,int Maximum=360,
            int RadSmall=30,int RadBig=45,int SM=AUTOMATIC,
            int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetRadiusSmall(Window,GC,GC,int);
    void    SetRadiusBig(Window,GC,GC,int);
    void    SetCenter(Window,GC,GC,int,int);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.5 Classe Horizontal Slider

A.5.1 Visão Externa

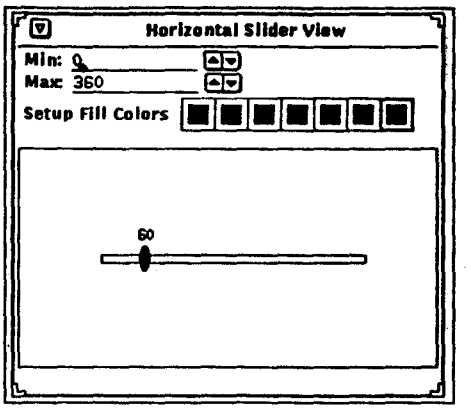

Figura A.4: A classe **Horizontal Slider**

A.5.2 Especificação

```
// HorSlider.h

#ifndef HORIZONTALSLIDER
#define HORIZONTALSLIDER

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

class HorizontalSlider : public NumericView {
private :
    int     Min;
    int     Max;
    int     Lenght;
```

```
    RectBox SliderTubeBox;
    RectBox SliderBox;
public :
    HorizontalSlider(int Minimum=0,int Maximum=360,
            int Len=120,int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.6 Classe Horizontal Tube

A.6.1 Visão Externa

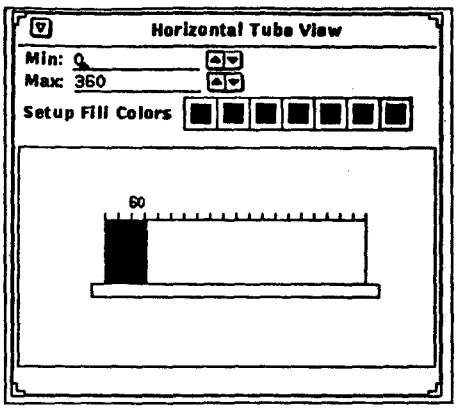

Figura A.5: A classe **Horizontal Tube**

A.6.2 Especificação

```
// HorTube.h

#ifndef HORIZONTALTUBE
#define HORIZONTALTUBE

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

class HorizontalTube : public NumericView {
private :
    int     Min;
    int     Max;
    int     Lenght;
```

```
    RectBox HorTubeBox;
    RectBox FootBox;
    RectBox TubeBox;
public :
    HorizontalTube(int Minimum=0,int Maximum=360,int Len=120,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.7 Classe Numeric

A.7.1 Visão Externa

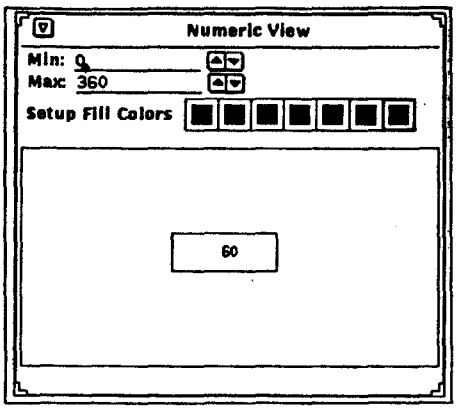

Figura A.6: A classe **Numeric**

A.7.2 Especificação

```
// Numeric.h

#ifndef NUMERIC
#define NUMERIC

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

#include "Relation.h"

class Numeric : public NumericView {
private :
    RectBox NumericBox;
```

```
public :
    Numeric(int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.8 Classe Pie

A.8.1 Visão Externa

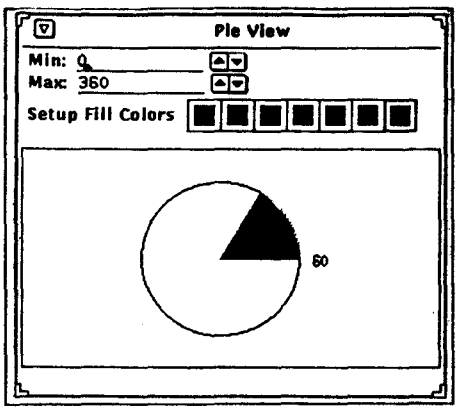

Figura A.7: A classe **Pie**

A.8.2 Especificação

```
// Pie.h

#ifndef PIE
#define PIE

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "Definition.h"
#include "NumericView.h"

class Pie : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     Radius;
    RectBox PieBox;
public :
    Pie(int Minimum=0,int Maximum=360,int Rad=45,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetRadius(Window,GC,GC,int);
    void    SetCenter(Window,GC,GC,int,int);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.9 Classe Scale

A.9.1 Visão Externa

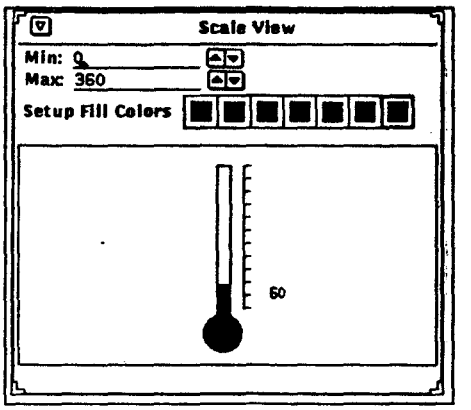

Figura A.8: A classe **Scale**

A.9.2 Especificação

```
// Scale.h

#ifndef SCALE
#define SCALE

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

class Scale : public NumericView {
private :
    int     Min;
    int     Max;
    int     Lenght;
```

```
    RectBox ScaleTubeBox;
    RectBox ScaleBox;
public :
    Scale(int Minimum=0,int Maximum=360,int Len=120,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.10 Classe Vertical Meter

A.10.1 Visão Externa

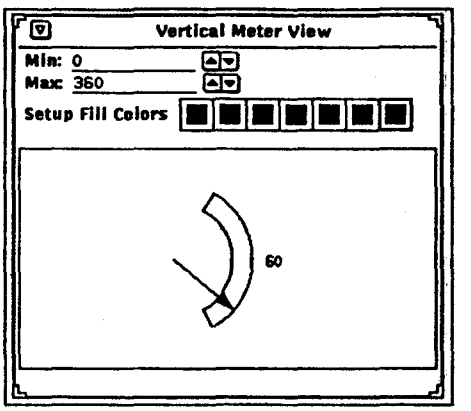

Figura A.9: A classe **Vertical Meter**

A.10.2 Especificação

```
// VerMeter.h

#ifndef VERTICALMETER
#define VETICALMETER

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "Definition.h"
#include "NumericView.h"

class VerticalMeter : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     RadiusSmall;
    int     RadiusBig;
    RectBox VerMeterBox;
public :
    VerticalMeter(int Minimum=0,int Maximum=360,
            int RadSmall=30,int RadBig=45,int SM=AUTOMATIC,
            int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetRadiusSmall(Window,GC,GC,int);
    void    SetRadiusBig(Window,GC,GC,int);
    void    SetCenter(Window,GC,GC,int,int);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.11 Classe Vertical Slider

A.11.1 Visão Externa

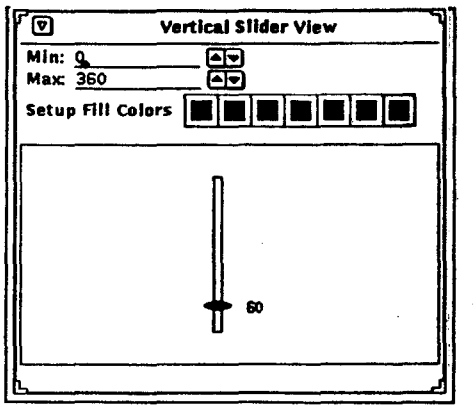

Figura A.10: A classe **Vertical Slider**

A.11.2 Especificação

```
// VerticalSlider.h

#ifndef VERTICALSLIDER
#define VERTICALSLIDER

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

class VerticalSlider : public NumericView {
private :
    int     Min;
    int     Max;
    int     Lenght;
```

```
    RectBox SliderTubeBox;
    RectBox SliderBox;
public :
    VerticalSlider(int Minimum=0,int Maximum=360,int Len=120,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.12 Classe Vertical Tube

A.12.1 Visão Externa

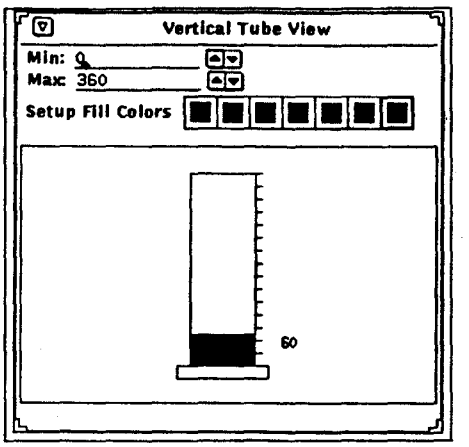

Figura A.11: A classe **Vertical Tube**

A.12.2 Especificação

```
// VerTube.h

#ifndef VERTICALTUBE
#define VERTICALTUBE

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <stdio.h>
#include <string.h>

#include "Definition.h"
#include "NumericView.h"

class VerticalTube : public NumericView {
private :
    int     Min;
    int     Max;
```

```
    int     Lenght;
    RectBox VerTubeBox;
    RectBox FootBox;
    RectBox TubeBox;
public :
    VerticalTube(int Minimum=0,int Maximum=360,int Len=120,
            int SM=AUTOMATIC,int R=0,int C=0);
    void    SetCoords(Point &);
    void    GetSize(Point &);
    void    Draw(Window,GC,GC);
    void    SetMin(Window,GC,GC,int);
    void    SetMax(Window,GC,GC,int);
    void    ShowValue(Window,GC,GC,int);
    int     Get(int,int);
    boolean InsideBox(int,int);
};

#endif
```

A.13 Classe DataRepresentation

A.13.1 Especificação

```
// DataRep.h

#ifndef DATAREP
#define DATAREP

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/canvas.h>
#include <xview/icon.h>
#include <xview/cms.h>
#include <xview/svrimage.h>
#include <xview/rect.h>

#include "Definition.h"
#include "Functions.h"
#include "ColorDef.h"

#include "Collection.h"
#include "NumericView.h"

class DataRep : public Collection {
    int    Min;
    int    Max;
    int    Val;
    int    OldValue;
    Point  NextPos;     // for automatic storage of next
                        // position to fit the figure
    int    MaxHeight;   // for automatic storage of maximum
                        // height of a figure
    Frame  ViewFrame;
    Canvas ViewCanvas;
    Window ViewWindow;
    GC     GcOutline;
```

```
    GC      GcFill;
public :
    DataRep(char *,int,int,int,int,int,int,int);
    ~DataRep();
    void    InsertNewView(NumericView *);
    void    UpdateNextPos(NumericView *);
    int     GetMin(void) { return Min; }
    int     GetMax(void) { return Max; }
    int     GetVal(void) { return Val; }
    Frame   GetViewFrame(void) { return ViewFrame; }
    GC      GetGcFill(void) { return GcFill; }
    void    ChangeMin(int);
    void    ChangeMax(int);
    void    ChangeVal(int,boolean);
    int     RequestNewData(int,int);
    void    UpdateChange(void);
    void    RejectChange(void);
    void    ReDraw(void);
    boolean InsideFrame(Window);
};

#endif
```

A.14 Classe Collection

A.14.1 Especificação

```
// Collection.h

#ifndef COLLECTION
#define COLLECTION

#include "Definition.h"

class Collection {
    typedef struct Node {
        void *Object;
        Node *Next;
    } Elem;
    Elem *Header, *Current, *Previous;
protected:
    void Put(void *);
    void *Start(void);
    void *Succ(void);
    void *LookSucc(void);
public:
    Collection() { Header = Current = Previous = NULL; }
    ~Collection();
    void Reset() { Header = Current = Previous = NULL; }
    void Remove(void);
    void ResetCurrent(void) { Previous = NULL;
                              Current = Header; }
};

#endif
```

A.15 Classe Relation

A.15.1 Especificação

```
// Relation.h

#ifndef RELATION
#define RELATION

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <xview/xview.h>
#include <xview/frame.h>

#include "Collection.h"
#include "DataRep.h"

typedef int (*FuncPoint)();

class Relation : public Collection {
    typedef struct _RelationPoint {
        DataRep   *View;
        FuncPoint  FuncAdd;
    } RelationPoint;
protected :
    void           UpdateViews(void);
public :
    ~Relation();
    void           Insert(DataRep *,FuncPoint);
    DataRep       *InputData(Window,int,int,int &);
    void           Refresh(void);
    boolean        TryToUpdateViews(void);
    DataRep       *GetDataRepFromFrame(Frame);
    RelationPoint *First(void)
                      { return (RelationPoint *)Start(); }
    RelationPoint *Next(void)
                      { return (RelationPoint *)Succ();  }
};

#endif
```

Apêndice B

Um exemplo prático

B.1 Preâmbulo

O exemplo através do qual será ilustrado a incoporação do princípio reativo em aplicações é um **simulador de circuitos lógicos orientado a objetos.**

Uma primeira versão do simulador de circuitos implementado foi apresentada durante o curso de **programação orientada a objetos** na VII Escola de Computação em julho de 1990 em São Paulo. Um especificação bem mais detalhada do **núcleo** do simulador está publicada em [TLX90].

Um simulador lógico é uma ferramenta de software utilizada para validar projetos de circuitos em um nível de abstração onde os sinais elétricos são representados por valores discretos. Os circuitos simulados são compostos de componentes lógicos interconectados, onde uma determinada saída de um componente alimenta uma ou mais entradas do próprio ou de outros componentes. A nível lógico é interessante saber, para um dado circuito, o estado das entradas e das saídas das portas lógicas que o compõem. Este estado é descrito em termos de dois níveis lógicos: o nível H (ou *high*) e o nível L (ou *low*). No modelo utilizado para a simulação ainda é utilizado um terceiro nível, denominado X, que representa o **estado indeterminado** durante uma transição de H para L e vice-versa ou representa um estado ainda indefinido por ocasião do início do processo de simulação. No modelo utilizado, uma entrada ou saída permanece no estado X, durante uma destas transições, por pelo menos uma unidade de tempo. E, um componente só reage a uma mudança no seu estado interno, em um determinado instante da simulação, uma unidade de tempo após a ocorrência da mudança do nível lógico em pelo menos uma de suas entradas - **atraso unitário.**

Os circuitos lógicos mais simples são formados por portas lógicas tais como portas NOT, AND, NAND, OR, NOR, XOR, EQV; por *flip-flops* tais como SR, JK e D; por injetores que alimentam o circuito sendo projetado: VDD, GND, ALEATORY, PREDEFINED e CLOCK; e por pontos de prova utilizados para monitoração dos resultados: PROBE.

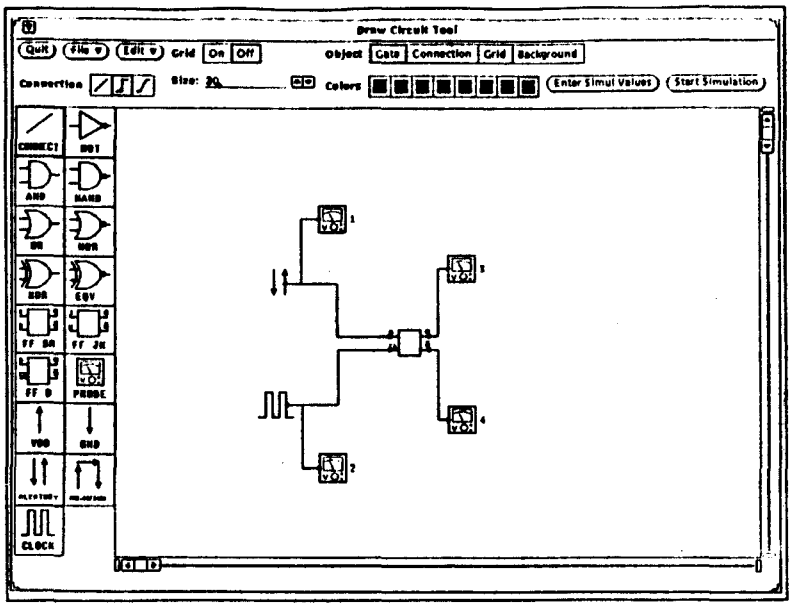

Figura B.1: Criação de um circuito lógico simples, com injetores e pontos de prova

Para exemplificar uma simulação lógica com atraso unitário, supõe-se que o circuito a ser simulado é somente um *flip-flop* do tipo D cujas entradas são alimentadas por um trem de pulsos aleatório (componente ALEATORY) na entrada 1 (na entrada referente ao dado D) e, por um trem de pulsos sincronizado (componente CLOCK) na entrada 2 (na entrada referente ao CK).

A figura B.1 ilustra o circuito construído no **simulador lógico implementado** e, a figura B.2 ilustra o resultado da simulação obtido para o FLIP-FLOP D.

B.2 Decomposição do problema em objetos

B.2.1 A classe descritiva dos circuitos lógicos

Como dito no capítulo 4, o primeiro passo para criação do modelo de objetos de uma aplicação qualquer e em particular deste editor/simulador é a identificação de quais os objetos que serão manipulados no contexto da simulação e quais as operações que serão necessárias aplicar sobre eles.

Claramente é necessário representar os **componentes básicos** e **circuitos lógicos** em termos de uma **coleção de componentes**, a partir dos quais o circuito será construído, e em termos de uma coleção de **especificações de interconexões**, onde cada

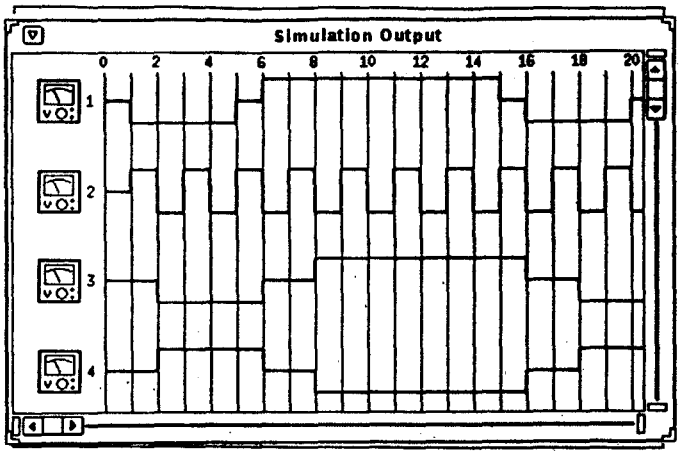

Figura B.2: Resultado da simulação de 50 trens de pulso do circuito ilustrado na figura B.1

especificação de interconexão (comumente chamada de sinal) relaciona uma dada saída de um componente com as entradas às quais deve ser conectada.

Em termos de circuito lógico, necessitamos prover entre seus métodos, operações de **especificação**, de **acionamento**, de **consulta** e, eventualmente, de **reinicialização**. Entre as operações de **especificação**, é possível listar as de:

- inclusão de componentes constituintes no circuito sendo especificado;
- criação de novos sinais a serem incluídos na coleção de sinais, explicitando para cada sinal criado a saída de componente onde se dará a origem do sinal propriamente dito;
- inclusão de entradas de componentes em um sinal (entradas estas a serem alimentadas pela saída onde se origina o sinal).

As operações de **acionamento** consistem basicamente em:

- avanço de uma unidade de tempo no relógio utilizado pelo simulador;
- propagação das saídas avaliadas no instante anterior seguida da reação dos componentes às novas entradas, provocando, eventualmente, uma alteração no estado das saídas dos componentes.

As operações de **consultas** provêem o usuário, entre outros, com informações sobre o estado dos pontos de prova monitorados durante a simulação e sobre a definição dos sinais, por exemplo.

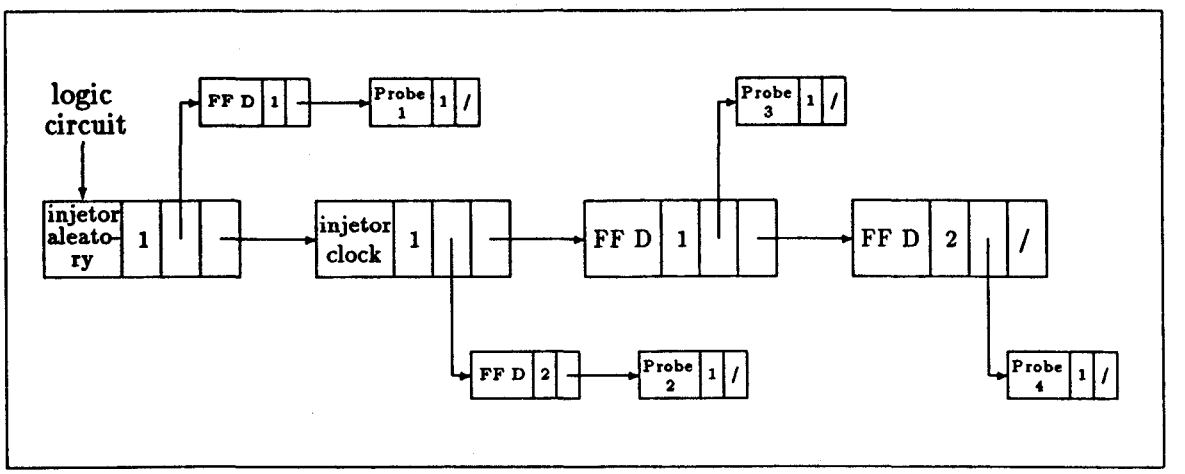

Figura B.3: Estrutura de dados correpondente ao circuit ilustrado na figura B.1

A criação de um circuito como o mostrado na figura B.1, no que se refere à interface, é bastante simples e intuitiva. O usuário "escolhe" os componentes que deseja na formação do circuito com auxílio do *menu* de componentes disponível e, através do *mouse*, aponta para a localização na área de *paint* onde o componente selecionado ficará posicionado. Após a criação de todos os componentes, o próximo passo é a criação das interconexões entre as entradas e saídas das portas lógicas. Isto é feito através do **objeto** CONNECT disponível no próprio painel de portas lógicas.

Paralelamente à criação e interconexão dos componentes, o núcleo do simulador vai **montando uma estrutura de dados** que deverá refletir o contexto do circuito sendo criado. Ao final da criação do circuito da figura B.1, por exemplo, uma possível estrutura de dados que corresponde ao circuito criado está ilustrada na figura B.3

É possível, analizando-se apenas esta estrutura de dados, recuperar o circuito projetado. A análise da estrutura deve ser iniciada a partir do objeto `LogicCircuit`. A análise é a seguinte: o injetor ALEATORY, através da sua saída de sinais número 1 está conectado ao FLIP-FLOP D através de sua entrada 1 e ao voltímetro PROBE 1 também a partir de sua primeira entrada. A próxima interconexão relaciona o injetor CLOCK que, a partir de sua saída 1 está conectado na entrada 2 do FLIP-FLOP D e na entrada 1 do voltímetro PROBE 2. A terceira interconexão relaciona a ligação entre a saída 1 do FLIP-FLOP D e a entrada 1 do voltímetro PROBE 3. E, a última ligação do circuito da figura B.1, é a que relaciona a saída 2 do FLIP-FLOP D à entrada 1 do ponto de prova PROBE 4.

O mapeamento de um circuito lógico na estrutura de dados implementada, como esperado, deve evidenciar apenas quais os componentes representados e as interconexões entre

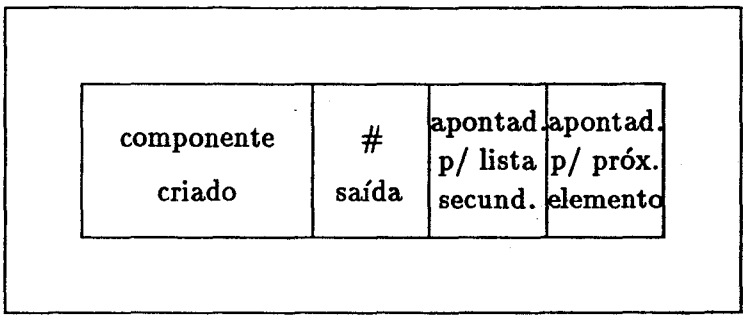

Figura B.4: Estrutura de um nó da lista principal da estrutura de dados que implementa um circuito lógico

eles. A estrutura implementada e ilustrada na figura B.3 pode ser decomposta, para efeito ilustrativo, em duas listas básicas. A primeira delas é a que é chamada de lista central ou principal. Esta lista relaciona os componentes lógicos básicos com as suas saídas ou simplesmente os componentes que originam os sinais produzidos. A estrutura de cada nó desta lista principal está representada na figura B.4.

A simplificação que a utilização desta estrutura induz é que apenas os componentes cujas saídas devem ser propagadas é que estarão representadas nesta estrutura de dados. Componentes cujas saídas não estejam conectados **não estarão** presentes na representação do circuito.

A segunda lista implementada e ilustrada na figura é a que representa as entradas dos componentes ligados as portas mapeadas pela primeira lista. Ou seja, é o destino ou para onde os sinais produzidos na primeira lista devem ser propagados. A estrutura de cada nó desta segunda lista está ilustrada na figura B.5.

Através desta lista secundária, é possível dizer, por exemplo, que para o circuito da figura B.1 mapeado na figura B.3, a saída 1 do injetor CLOCK, deve ser propagada no mesmo instante de simulação ao componente FLIP-FLOP D através da entrada 2 e ao ponto de prova PROBE 2 através de sua entrada 1. Ou ainda que a saída 2 do componente FLIP-FLOP D deve ser propagada à entrada 1 do componente ponto de prova PROBE 4.

Para iniciar a simulação de um circuito, uma vez definida e montada esta estrutura de dados, basta apenas que um método da classe `LogicCircuit` definida "percorra" toda esta estrutura a quantidade de vezes que o usuário selecionar para a simulação, avançando o relógio utilizado para simulação e propagando os resultados das portas interconectadas.

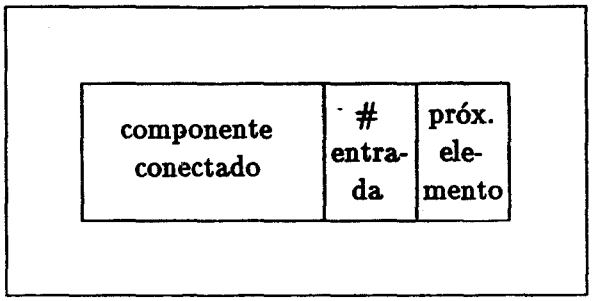

Figura B.5: Estrutura de um nó da lista secundária da estrutura de dados que implementa um circuito lógico

B.2.2 A hierarquia de componentes

Os componentes implementados que constituem as peças formadoras de um circuito lógico, como dito, são componentes simples tais como portas NOT, AND, NAND, XOR, EQV, etc.

A sua saída é função apenas dos valores registrados nas entradas (eventualmente, no caso dos *flip-flops*, o estado anterior é requerido para "cálculo" do próximo estado). Aqui, novamente são encontradas semelhanças entre os diversos componentes. Estas semelhanças podem ser fatoradas e utilizadas para a criação de uma classe abstrata, cujo código é então compartilhado pelas classes especializadas a partir da classe mais abstrata.

No caso de componentes, foi definida a classe básica `Gate` da qual são derivadas as portas lógicas específicas. Na definição desta hierarquia de classes foi novamente utilizada a técnica de fatoração implementando-se na classe básica as características mais gerais aplicáveis a componentes e nas classes derivadas as propriedades específicas de cada porta lógica.

Para especificação da classe `Gate`, por exemplo, poderiam ser fatoradas propriedades tais como:

- a coordenada cartesiana do componente sendo criado;
- o seu tipo e grupo (injetor, ponto de prova, componente simples, ...);
- o número de entradas e saídas que possui;
- a cor adotada na sua criação;
- um apontador para onde estão definidos e representados os valores das entradas e saídas do componente sendo criado.

Enquanto que, as operações básicas definidas neste nível poderão ser:

```
// And.h
#ifndef AND
#define AND
#include "Gate.h"
...
class And :  public Gate {
public:
   And(int X,int Y,unsigned long Pixel);
   void OutputUpdate(void);
   void Draw(Window,GC,int,int);
   ...
};
#endif
```

```
void And ::  OutputUpdate(void)
{
   if (Input[0] == HIGH && Input[1] == HIGH)
      Output[0] = HIGH;
   else
      if (Input[0] == LOW || Input[1] == LOW)
         Output[0] = LOW;
      else
         Output[0] = UNDEFINED;
}
void And ::  Draw(Window PaintWindow,...)
{
   // code for draw
{
```

Figura B.6: Parte da especificação e da implementação da classe **And**

- inicializar as entradas e saídas;
- recuperar as coordenadas cartesianas do componente;
- recuperar o valor de uma entrada específica;
- definição dos métodos virtuais **Draw** e **OutputUpdate**, com implementação **nula**, para tornar possível o **polimorfismo** entre os componentes lógicos criados.

Uma vez definida a classe abstrata de componentes lógicos, será necessário definir as operações específicas de cada componente (a definição das subclasses da classe **Gate**). A figura B.6 ilustra uma parte da especificação e da implementação de uma subclasse de **Gate** (a classe **And**). É interessante observar que dentre as operações específicas que são implementadas nesta classe, estão especificadas a operação que "desenha" (o método **Draw**) o componente e a que implementa a **tabela verdade** da porta sendo especificada (o método **OutputUpdate**).

A hierarquia final de componentes que implementam o simulador de circuitos lógicos é mostrada na figura B.7. As classes **Net**, **NetList**, **Components** e **Collection** implementam operações abstratas sobre listas com encadeamento simples e operações sobre a estrutura de dados ilustrada na figura B.3. Os segmentos de reta, como dito no capítulo 4[1], representam as operações abstratas de especialização e generalização; os segmentos tracejados, as operações de agregação e refinamento; e o segmento pontilhado, as de classificação e instanciação.

[1] ver subseção 5.1.1.

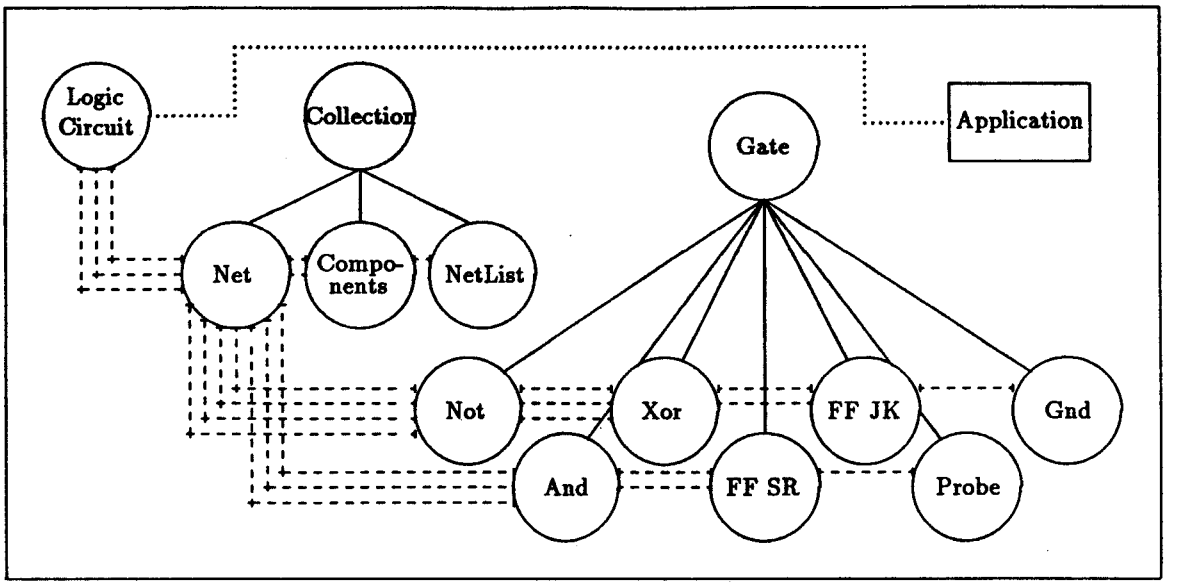

Figura B.7: Hierarquia de abstrações do simulador de circuitos lógicos

B.3 Ferramentas adicionais

O simulador de circuitos implementado possui algumas ferramentas construídas com o propósito de que houvesse simplificação na construção de um circuito lógico. A interface utilizada seguiu o pardrão de especificação do XView e conseqüêntemente do padrão OPENLOOK. Conforme mostrado na figura B.1, a interface produzida possui uma série de botões, *menus*, *panel choices*, etc. Dentre as opções disponíveis nesta interface é interessante destacar as de:

- criação de uma "malha" para auxílio na edição e na escala dos componentes sendo criados (o *panel choice* `Grid`). É possível também dimensionar o tamanho desta malha através do objeto do painel chamado `Size` (objeto `PANEL_NUMERIC_TEXT`);
- seleção do objeto e da cor a ser utilizada na sua criação (opções `Object` e `Colors`);
- operações básicas (armazenamento, recuperação, cópia, remoção, ...) sobre circuitos sendo especificados. Estas opções estão encapsuladas no *menu pulldown* `File` mostrado na figura B.1. Para ilustração de uma destas opções, na figura B.8 é mostrada todo o "caminho" percorrido pelo usuário para ativação da criação (opção `Save`) ou atualização de um circuito em memória secundária (disco).

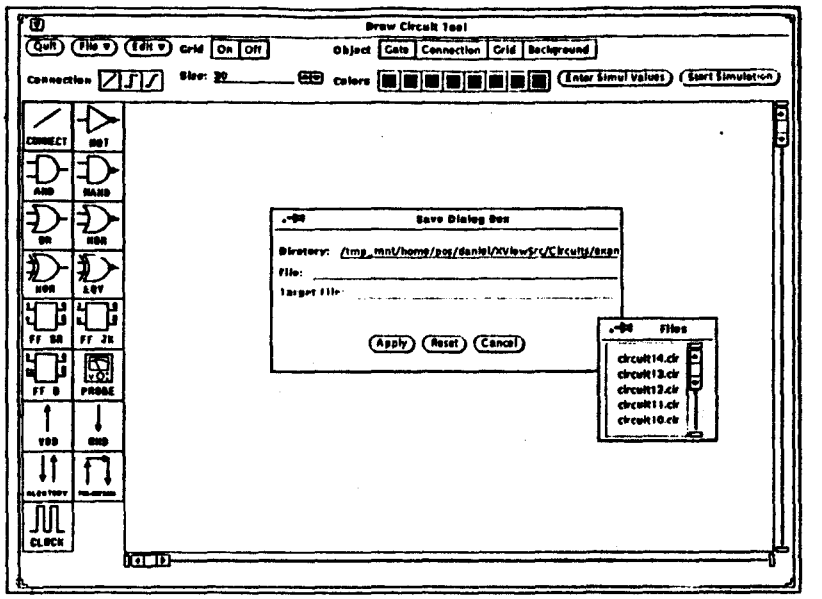

Figura B.8: Criação ou atualização de um circuito lógico em disco

B.4 O princípio reativo

O princípio reativo foi incorporado ao sistema através do botão **Enter Simulation Values** disponível no painel de opções da figura B.1. A especificação de quais componentes interativos farão parte do princípio reativo foi feita de maneira análoga à mostrada na figura 5.12. A incorporação do princípio reativo em uma aplicação como esta (simulador de circuitos lógicos) reforça a afirmação de que o princípio de múltiplas visões com reatividade pode ser incorporado facilmente a diversos tipos de aplicações. Para que isto ocorra basta que haja necessidade de interação com o usuário. O princípio reativo produzido para o simulador de circuitos lógicos implementado está ilustrado na figura B.9.

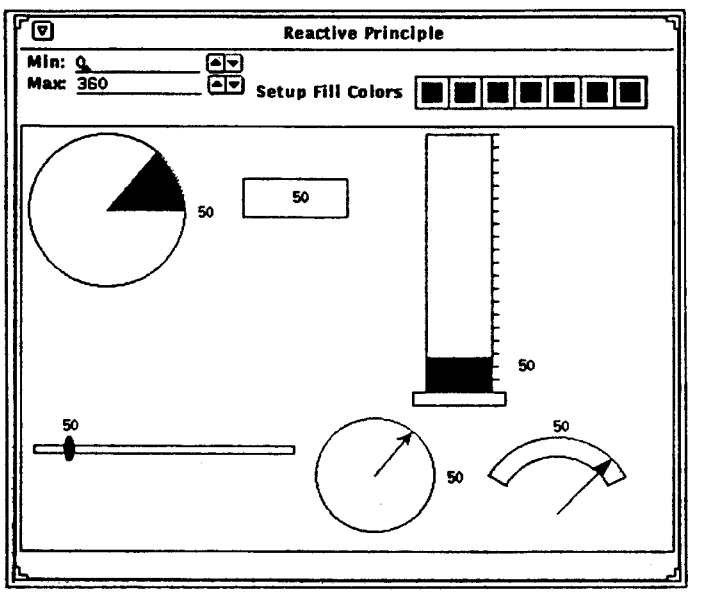

Figura B.9: Princípio reativo produzido para o simulador lógico

Índice

T

V

X

Bibliografia

[Aye89] Kenneth E. Ayers. An object-oriented logic simulator. *Dr. Dobb's Journal*, pages 72–78, December 1989.

[B+91] Gordon Blair et al. *Object-Oriented Languages, Systems and Applications.* Pitman Publishing, University of Lancaster, London, 1991.

[Ber89] John Berry. *The Wait Group's C++ Programming.* Howard W. Sams & Company, 1989.

[CCM87] Lisa A. Call, David L. Cohrs, and Barton P. Miller. CLAM – an open system for graphical user interfaces. In *Proceedings OOPSLA'87*, pages 277–286. Sigplan Notices, October 1987.

[CH86] Brad Cox and Bill Hunt. Objects, icons and software-ics. *Byte*, pages 161–176, August 1986.

[Cou87] Joelle Coutaz. The construction of user interfaces and the object paradigm. In *Proceedings ECOOP'87*, pages 121–130, 1987.

[DHM89] Mahesh H. Dodani, Charles E. Hughes, and Michael Moshell. Separation of powers. *Byte*, pages 255–262, March 1989.

[Duf86] Charles B. Duff. Design an efficient language. *Byte*, pages 133–139, August 1986.

[Fei87] Peter Feiler. User interface technology survey. Technical Report CMU/SEI-87-TR-6, Carnegie-Mellon University, Pittsburgh - Pensylvania, April 1987.

[Fiu89] Eugene Fiume. Active objects in the construction of graphical user interfaces. *Computer & Graphics*, 13(3):321–327, March 1989.

[GE87] Mark Grossman and Raimund K. Ege. Logical composition of object-oriented interfaces. In *Proceedings OOPSLA'87*, pages 295–306. Sigplan Notices, October 1987.

[GF87] G. M. Gwei and E. Foxley. A flexible synonym interface with application examples in CAL and help environments. *The Computer Journal*, 30(6):551–557, June 1987.

[Goo89] Mark Goodwin. *User Interfaces in C++ and Object-Oriented Environment.* MIS Press, Portland – Oregon, 1989.

[GW90a] Erich Gamma and André Weinand. *ET++ - A portable C++ class library for a Unix environment.* OOPSLA - ECOOP, October 1990.

[GW90b] Erich Gamma and André Weinand. *ET++ - Introdution and Installation.* Union Bank of Switzerland, November 1990.

[Har87] Steven Harrington. *Computer Graphics (a programming approach).* McGraw-Hill Book Company, second edition, 1987.

[Hel90] Dan Heller. *XView Programming Manual*, volume seven. O'Reilly & Associates, Inc., second edition, July 1990.

[HH89] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems for its management. Technical Report 1, Virginia Polytechnic Institute and State University, Blacksburg - Virginia, March 1989. Vol. 21.

[Hu90] David Hu. *Object-Oriented Environment in C++, a user-friendly interface.* MIS Press, Portland – Oregon, 1990.

[Jon89] Oliver Jones. *Introduction to the X Window System.* Prentice-Hall, 1989.

[Koe89] Andrew Koening. How virtual functions work. *Journal of Object Oriented Programming*, pages 73–74, January/February 1989.

[Kow83] Tomasz Kowaltowski. *Implementação de linguagens de programação.* Editora Guanabara Dois, Rio de Janeiro, RJ, Janeiro 1983.

[KR78] Brian W. Kernigan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, New Jersey - NJ, 1978.

[Lam86] Leslie Lamport. LaTeX – *A Documment Preparation System.* Addison-Wesley Publishing Company, Digital Equipament Corporation, 1986.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ graphical interface toolkit. In *Proceedings of USENIX C++ Workshop*, Santa Fé - New Mexico, 1987.

[Mic89a] Sun Microsystems. *AT&T C++ Language System - Reference Manual.* Sun Microsystems, 1989.

[Mic89b] Sun Microsystems. *AT&T C++ Language System - Select Readings.* Sun Microsystems, 1989.

[Mic89c] Sun Microsystems. *AT&T C++ Translator Library.* Sun Microsystems, 1989.

[Mic89d] Sun Microsystems. *Sun C++ Programmer's Guide.* Sun Microsystems, 1989.

[Mic90] Sun Microsystems. *OpenLook Graphical User Interface Function Specification.* Sun Microsystems, July 1990.

[MSB90] John Alan McDonald, Wener Stuetzle, and Andreas Buja. Painting multiple views of complex objects. *ECOOP/OOPSLA'90*, pages 245–257, October 1990.

[Mul89] Mark Mullin. *Object-Oriented Programming Design, with Examples in C++.* Addison-Wesley Publishing, 1989.

[Nyg86] Kristen Nygaard. Basics concepts in object oriented programming. *SIGPLAN Notices*, 21(10):128–132, October 1986.

[O'D85] John T. O'Donnell. Dialogues: A basis for constructiong programming environments. *Communications ACM*, 0(6):19–27, June 1985.

[Pas86] Geoffrey A. Pascoe. Elements of object-oriented programming. *Byte*, pages 15–20, August 1986.

[Pou86] Dick Pountain. Object-oriented forth. *Byte*, pages 227–233, August 1986.

[Pur85] James Purtilo. Polylith: An environment to support management of tool interfaces. *Communications ACM*, 0(6):12–18, June 1985.

[Rei86] Steven P. Reiss. An object-oriented framework for graphical programming. *SIGPLAN Notices*, 21(10):49–57, October 1986.

[Sch86a] Kurt Schmucker. Macapp: An application framework. *Byte*, pages 189–193, August 1986.

[Sch86b] Kurt Schmucker. Object-oriented languages for the macintosh. *Byte*, pages 177–185, August 1986.

[Str86] Bjarne Stroustrup. An overview of C++. *SIGPLAN Notices*, 21(10):7–18, October 1986.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, pages 10–20, May 1988.

[Tak89] Tadao Takahashi. *O Paradigma de Objetos: Introdução e Tendências.* VIII Jornada de Atualização em Informática. Projeto ETHOS, Uberlândia–MG, Junho 1989.

[Tes86] Larry Tesler. Programming experiences. *Byte*, pages 195–206, August 1986.

[Tho89] Dave Thomas. What's in an object? *Byte*, pages 231–240, March 1989.

[TLX90] Tadao Takahashi, Hans Liesenberg, and Daniel Xavier. *Introdução a Programação Orientada a Objetos, uma visão integrada do paradigma de objetos.* VII Escola de Computação. IME–USP, São Paulo–SP, Julho 1990.

[W+88] André Weinand et al. ET++ – an object-oriented application framework in C++. In *Proceedings OOPSLA'88*, pages 46–57. Sigplan Notices, September 1988.

[Weg89] Peter Wegner. Learning the language. *Byte*, pages 245–253, March 1989.

[Wei89] André Weinand. Design and implementation of ET++, a samless object-oriented application framework. *Structured Programming*, 10(2):63–87, June 1989. Spring–Verlag.

[Xav90] Daniel Tavares Correia Xavier. Extensões no ET++ para suportar o princípio de múltiplas visões com reatividade. Universidade Estadual de Campinas - UNICAMP, Departamento de Ciência da Computação, Pré-defesa de tese de mestrado, Outubro 1990.