UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Ambiente Visual para Programação Distribuída em Java

por

JULIANO MALACARNE

Dissertação submetida à avaliação como requisito
parcial para a obtenção do grau de Mestre em
Ciência da Computação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, fevereiro de 2001.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Malacarne, Juliano

Ambiente Visual para Programação Distribuída em Java / por Juliano Malacarne. - Porto Alegre: PPGC da UFRGS, 2001.

155 p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientador: Geyer, Cláudio F. R.

1. Ambiente de Programação. 2. Programação Visual. 3. Programação Distribuída. 4. Objetos Distribuídos. 5. Java. I. Geyer, Cláudio Fernando Resin. II. Título.

Agradecimentos

Agradeço a Deus pela companhia e por tudo de bom que tem me proporcionado.

Agradeço à minha família, formada pelo meu pai Nelso, pela minha mãe Lourdes e pela minha irmã Luciana, por todo o amor e carinho que nunca deixaram de me dar.

Agradeço a todos os colegas e amigos que acrescentaram ao tempo de estudo e trabalho momentos de alegria, divertimento e aprendizado de vida.

Agradeço aos professores da UFRGS, em especial ao professor orientador Geyer, pelos conhecimentos transmitidos ao longo do curso de graduação, de mestrado e de todas as atividades que desenvolvi nesta universidade.

Agradeço ao CNPq, pelo suporte financeiro durante o mestrado e as atividades de iniciação científica e pelo grande incentivo que tem dado à pesquisa neste país. Esse apoio é imprescindível para que o Brasil se desenvolva e resolva os problemas de sua população.

Sumário

Lista de Abreviaturas

BDK	*Beans Development Kit*
CBSE	*Component-Based Software Engineering*
CC++	*Compositional C++*
CORBA	*Common Object Request Broker Architecture*
DCOM	*Distributed Component Object Model*
DII	*Dynamic Invocation Interface*
DSM	*Distributed Shared Memory*
GIOP	*General Inter Orb Protocol*
GPPD-II	*Grupo de Processamento Paralelo e Distribuído do Instituto de Informática*
GRADE	*Graphical Application Development Environment*
HetNOS	*Heterogeneous Network Operating System*
HTTP	*Hyper Text Transfer Protocol*
IDL	*Interface Definition Language*
IIOP	*Internet Inter Orb Protocol*
JIPE	*Java Integrated Programming Environment*
JNDI	*Java Naming and Directory Interface*
MPI	*Message Passing Interface*
OMG	*Object Management Group*
POOL	*Parallel Object Oriented Language*
P-RIO	*Parallel Reconfigurable Interconnectable Objects*
PVM	*Parallel Virtual Machine*
RAD	*Rapid Application Development*
ReMMoS	*Replication Model in Mobility Systems*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
SII	*Static Invocation Interface*
SMTP	*Simple Mail Transfer Protocol*
SR	*Synchronizing Resources*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
URL	*Uniform Resource Location*
VDIR	*Voyager Federated Directory Service*
VPE	*Visual Programming Environment*
WWW	*World Wide Web*

Lista de Figuras

Lista de Tabelas

Resumo

Em vista da maior complexidade da programação paralela e distribuída em relação à programação de ambientes centralizados, novas ferramentas vêm sendo construídas com o objetivo de auxiliar o programador desses ambientes a desempenhar sua tarefa de formas mais eficazes e produtivas. Uma das ferramentas que há algum tempo tem sido usada na programação centralizada e aos poucos está sendo empregada também na programação concorrente é a programação visual.

A programação visual se vale da presença de elementos visuais na especificação dos programas como peças chaves do processo de desenvolvimento de *software*. No caso específico da programação concorrente, a programação visual é especialmente útil pela capacidade que os gráficos têm de representar de forma mais adequada estruturas bidimensionais. Um programa concorrente, por relacionar no espaço diversos elementos com seus próprios fluxos de execução, faz surgir duas dimensões de análise que são mais difíceis de serem observadas através de programas textuais.

Atualmente existem ferramentas de programação visual paralela e distribuída, mas a ênfase é dada na programação paralela, sem muita atenção a aplicações de sistemas abertos ou cliente-servidor. Além disso, tais ferramentas sofrem da falta de apoio à engenharia do *software*. Considerando essas deficiências, este trabalho apresenta uma ferramenta de programação visual para o desenvolvimento de aplicações compostas por objetos distribuídos que ofereça também a possibilidade de aplicar os principais conceitos da engenharia de *software*, como reutilização e orientação a objeto. Nesta ferramenta, o programador especifica de maneira visual a estrutura do seu programa, insere o código textual para a lógica da aplicação e o ambiente se encarrega do tratamento da distribuição e da comunicação de mais baixo nível. A aplicação é representada como um grafo dirigido, onde os nodos representam os objetos distribuídos e os arcos indicam os relacionamentos existentes entre esses objetos. A especificação dos programas é modular, baseando-se na reunião de componentes reutilizáveis, o que torna o sistema altamente configurável e extensível.

Tanto a implementação da ferramenta quanto o código das aplicações geradas usam a linguagem de programação Java. A linguagem de programação visual projetada não especifica detalhes a respeito de como irá funcionar a comunicação e distribuição dos objetos. Portanto, foram implementados componentes para comunicação e outros recursos de programação distribuída, como *locks* e dados globais para serem usados nas aplicações. Para validar os principais objetivos da ferramenta, foram implementados alguns exemplos de aplicações distribuídas, como um pequeno sistema de bate-papo.

Palavras-chave: Ambiente de Programação, Programação Visual, Programação Distribuída, Objetos Distribuídos, Java.

TITLE: "A VISUAL PROGRAMMING ENVIRONMENT FOR DISTRIBUTED PROGRAMMING IN JAVA"

Abstract

Due to the bigger complexity of the distributed and parallel systems compared to centralized systems, new tools have been built to help the programmer of these environments to do his/her task in a more productive and effective way. One of the tools that has been used in the centralized programming and also in the concurrent programming is the visual programming.

The visual programming uses visual elements as key parts of the software development process to specify programs. In the concurrent programming, the visual programming is especially useful due to the power that the graphics have to represent bidimensional structures more properly. A concurrent program, which binds in the space several elements to their own flows of execution, creates two dimensions of analysis that are harder to be observed through textual programs.

Nowadays, there are some visual programming tools for parallel and distributed systems, but the emphasis is given to parallel programming, with a little support to open and client-server systems. Besides, these tools depreciate software engineering support. Considering these deficiencies, this works presents a visual programming tool to develop applications composed by distributed objects that also offers the possibility to apply the more important software engineering concepts, like reuse and object orientation. In this tool, the programmer specifies his/her program structure visually and inserts the textual code for the application logic, while the environment treats the low-level communication and distribution processes. The application is represented as a directed graph, where the nodes represent the distributed objects and the edges indicate the relationships existing among these objects. The program specification is modular, based on the joining of reusable software components, which makes the system highly configurable and extensible.

Both the implementation of the tool and the code of the applications that are generated use the Java programming language. The visual programming language designed doesn’t specify details about the way the communication and the object distribution take place. So, some components were implemented for communication and other distributed programming resources, like locks and global data to be used in the applications. To validate the main objectives of the tool, it was implemented a little chat system, a typical client-server application with distributed objects.

Keywords: Programming Environment, Visual Programming, Distributed Programming, Distributed Objects, Java.

1 Introdução

Este trabalho visa ao projeto e à implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java. A programação é baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. Procurou-se com este trabalho desenvolver uma ferramenta que aproveitasse as melhores características das ferramentas visuais de programação paralela e distribuída e das ferramentas de programação visual em Java, aproveitando de cada uma, respectivamente, os recursos de visualização para a programação concorrente e as características de engenharia de *software*.

1.1 Motivação

A programação paralela e distribuída é reconhecidamente mais complexa do que a programação de ambientes centralizados. Várias são as razões para que isto aconteça: diversas tarefas independentes, comunicação, sincronização e temporização [CAI95]. Essas dificuldades prejudicam todo o ciclo de desenvolvimento dos programas, desde a fase de projeto até as fases de implementação e de depuração. Na fase de projeto devem ser construídos algoritmos que costumam ser difíceis de serem entendidos, testados e provados matematicamente. A implementação deve adequar os recursos disponíveis ao modelo teórico do projeto, o que nem sempre é uma tarefa trivial, como no caso da emulação de memória compartilhada numa rede de estações de trabalho. Por fim, a depuração sofre da falta de determinismo das execuções e do grande número de elementos envolvidos num único programa paralelo ou distribuído.

Por tudo isso, em se tratando de desenvolvimento de *software*, a quantidade e a qualidade das aplicações concorrentes têm deixado a desejar, principalmente se comparadas às aplicações centralizadas. Durante um certo tempo, onde a disponibilidade de redes de computadores e de *hardware* distribuído era restrita e a exigência por aplicações desse tipo não era tão grande, a limitação foi em parte aceita. Entretanto, o surgimento de novas aplicações distribuídas e a possibilidade de implementá-las em cima de *hardware* cada vez mais barato criou a necessidade de se aprimorar a qualidade do *software* empregado em tais plataformas. Para suprir essa necessidade, diversas ferramentas de apoio ao desenvolvimento têm surgido com características próprias para resolver diferentes tipos de problemas. No caso de aplicações paralelas e distribuídas, as ferramentas predominantes são as linguagens de programação textual, que, através de um modelo próprio que suporte a concorrência ou através de bibliotecas de programação, são capazes de expressar o paralelismo e a distribuição.

As linguagens de programação textual para aplicações paralelas e distribuídas são poderosas, mas tornam difíceis o desenvolvimento e a compreensão dos programas. A causa dessa dificuldade reside no fato de que um dos principais atributos dos programas concorrentes é a sua bidimensionalidade. Além de se considerarem as ações individuais de processos ou *threads*, devem ser levadas em conta também as interações entre eles, o que dá ao problema duas dimensões de análise. As linguagens textuais caracterizam-se pelas suas propriedades unidimensionais e por isso não são o tipo de representação mais apropriado aos programas concorrentes.

Com a finalidade de solucionar parte do problema e dar aos programas concorrentes uma representação mais natural e mais próxima do usuário do que da máquina, uma ferramenta usada há algum tempo na programação centralizada vem recebendo atenção. Tal ferramenta é a programação visual, que se caracteriza pela existência de representações visuais no desenvolvimento de programas [BUR95]. Essa ferramenta tem se mostrado útil na programação concorrente devido à capacidade que os gráficos têm de representar com maior clareza estruturas bidimensionais, possibilitando que relacionamentos entre diferentes entidades dispersas no espaço tenham uma descrição mais fácil de ser compreendida pelo ser humano.

Aproveitando esta vantagem da programação visual, muitas ferramentas foram criadas com a finalidade de superar parte das deficiências apresentadas pelas ferramentas existentes até então na programação de aplicações paralelas e distribuídas. Pode-se citar, entre outros, os ambientes de programação visual CODE [NEW92], VPE [NEW95], HeNCE [BEG93], VisualProg [SCH96][MAL97], GRADE [KAC97], P-RIO [LOQ98], Tracs [BAR95], Paralex [DAV96] e PVMBuilder [PED99]. A utilização de uma linguagem visual para a programação é conseqüência de uma diretriz comum a essas ferramentas: a facilidade de uso. Assim como na programação centralizada, a facilidade de uso tem sido buscada para aumentar a produtividade e a qualidade dos programas, aspectos críticos em programação concorrente. Os ambientes então acrescentam uma série de características para que isso seja possível: integração de ferramentas, reutilização de código, flexibilidade, eficiência e portabilidade das aplicações geradas. A partir da descrição visual do programa, as ferramentas geram código automaticamente e assim contribuem também para a diminuição dos erros que são muito comuns em programas concorrentes.

Apesar de a maioria dos ambientes visuais afirmarem o apoio ao desenvolvimento de aplicações distribuídas, pouco existe nessa área e a ênfase maior da programação visual se dá mesmo na programação paralela. Isso se verifica pelo suporte que as ferramentas oferecem à geração de código para as máquinas virtuais paralelas PVM [BEG91] e MPI [MPI94] e por outras características do modelo, como por exemplo a replicação de código para exploração do paralelismo de dados. Pouco existe para a conexão com servidores externos como HTTP ou SMTP e muito pouco é falado a respeito de conceitos mais próximos de sistemas distribuídos, como tolerância a falhas e aplicações cliente-servidor.

1.2 Contexto do trabalho

Juntamente com este trabalho estão sendo desenvolvidos outros projetos no GPPD-II UFRGS (Grupo de Processamento Paralelo e Distribuído do Instituto de Informática da UFRGS) com os quais haverá integração. Pretende-se com isto criar dentro do GPPD um grande projeto de ampla abrangência sobre objetos distribuídos.

Os trabalhos atuais que serão integrados são o ReMMoS (*Replication Model in Mobility Systems*) [FER2000] e uma ferramenta de visualização de programas em Java [ARA2000]. O primeiro deles tem por objetivo principal prover de forma transparente ao programador um ambiente de controle de réplicas de objetos com vistas ao aumento do desempenho das aplicações. O segundo possibilitará a visualização da execução e dos resultados das aplicações criadas com a ferramenta apresentada nesta dissertação. Com a visualização da execução será possível compreender melhor as aplicações distribuídas e obter assim programas mais confiáveis e com melhor desempenho.

Além desses dois, está em estudo a integração também do DEPAnalyzer [AZE2000], um analisador de dependências para programas seqüenciais orientados a objeto cuja finalidade é extrair informações a respeito da comunicação entre os objetos, oferecendo assim subsídios para o seu porte para um ambiente distribuído. Embora a ferramenta proposta aqui não seja de programação seqüencial, estão se procurando maneiras de se aproveitarem as informações providas pelo DEPAnalyzer e assim facilitar a programação das aplicações.

1.3 Modelo

Considerando a situação atual dos ambientes visuais, o objetivo deste trabalho é modelar e implementar uma ferramenta de programação visual que priorize o atendimento a requisitos mais importantes para a programação distribuída do que para a programação paralela. A este conjunto pertencem aplicações do tipo cliente-servidor e que interoperam entre sistemas criados por diferentes desenvolvedores, recursos praticamente inexistentes nas ferramentas de programação paralela. Além disso, assim como todo ambiente de programação, deve buscar a facilidade de uso e oferecer recursos ao usuário que o auxiliem no desenvolvimento do *software*: reutilização, modularização, extensibilidade e geração automática de código.

Com base nesses requisitos, diferentes soluções podem ser dadas ao problema. Uma das soluções é a integração da programação orientada a objeto com os sistemas distribuídos, dando origem ao modelo de objetos distribuídos. A orientação a objeto resolve boa parte dos problemas de reutilização, modularidade e encapsulamento desejados em quaisquer aplicações, principalmente nas de grande porte. A utilização de uma linguagem orientada a objeto é o meio natural através do qual as aplicações serão construídas nesta ferramenta.

Tomadas as decisões principais, deve-se partir para a definição da linguagem de programação visual, uma das partes mais importantes do projeto de um ambiente visual. Muitas das características desejáveis do ambiente estão sob a responsabilidade da sua linguagem visual. Na programação distribuída, uma aplicação do usuário é composta por um conjunto de objetos distribuídos. Optou-se nesta ferramenta pela adoção de um modelo onde tal aplicação é representada através de um grafo dirigido, onde os objetos distribuídos são representados por nodos e os relacionamentos entre eles são visualizados explicitamente por meio de arcos que conectam tais nodos. Há vários tipos de relacionamentos possíveis entre diferentes nodos, sendo que os principais são a invocação remota de métodos, a criação remota de objetos e a comunicação por meio de mensagens explícitas.

A especificação da aplicação contém diferentes tipos de objetos. Os objetos distribuídos propriamente ditos são os nodos do grafo no programa visual e executam fisicamente em diferentes espaços de memória. A comunicação, entretanto, é implementada por outros objetos acoplados a esses objetos distribuídos. Tais objetos especiais são os componentes de programação distribuída, representados principalmente pelas portas de comunicação e referências remotas para chamadas de métodos. Essa abordagem de componentes permite que a comunicação seja implementada de diversas maneiras, possibilitando ainda que seja alterada e personalizada de acordo com as necessidades de cada aplicação. Para tanto, basta que um novo tipo de componente seja implementado.

A comunicação entre o componente e o objeto que o contém é feita através de eventos. Todo o comportamento de um objeto é controlado pelos eventos que ele gera e

recebe. Por exemplo, quando um objeto possuir dados prontos para receber em uma determinada porta de comunicação (um componente declarado no objeto), ele será notificado do evento que dá conta dessa condição. Durante a fase de desenvolvimento, o programador definirá a ação a ser tomada em função da ocorrência do evento em questão. O emprego do modelo de eventos é útil também como forma de aumentar a concorrência, pois um objeto poderá realizar outras tarefas enquanto espera a mensagem, não tendo que executar sempre uma chamada de recebimento de mensagem que bloqueie o processamento antes do tempo necessário. A notificação de erros e falhas na comunicação também é facilitada através deste esquema.

O modelo de programação apresenta outros elementos além de objetos distribuídos e portas de comunicação. Entre outros, estão disponíveis também *locks*, dados globais à rede, portas de chamadas de serviço, serviços e servidores virtuais. Todos eles estão organizados numa hierarquia de objetos e definidos em termos de propriedades, métodos e eventos, seguindo o estilo de programação de muitas das ferramentas RAD [CAR95] de programação visual para a plataforma Windows [SCH98]. Esse tipo de definição faz com que os objetos da aplicação (chamados de componentes no contexto dessas ferramentas) sejam totalmente independentes entre si, tornando modular o desenvolvimento da aplicação. A possibilidade de extensão do ambiente é uma conseqüência natural da modularização, permitindo que novos componentes sejam adicionados à ferramenta conforme a necessidade.

Portanto, o usuário terá à sua disposição um modelo de programação orientado a eventos, com definição modular de componentes e com geração de código portável entre diferentes arquiteturas. Em vista dessas características, uma solução adequada para a implementação deste modelo pode ser obtida com a utilização de uma linguagem que seja orientada a objeto e também portável. Há um grande número de linguagens orientadas a objeto que poderiam ser usadas como interface de programação ao usuário da ferramenta. Entretanto, apenas uma parte restrita delas atende plenamente ao requisito da portabilidade. Muitas linguagens podem ser consideradas portáveis no sentido de que o código não precisa ser alterado entre diferentes plataformas de *hardware* e de *software*, mas elas sempre exigem a recompilação do código fonte. Java [SUN2000e], por não ser apenas uma linguagem de programação, mas sim um ambiente completo de desenvolvimento e execução, permite que mesmo um código já compilado possa ser executando sem quaisquer alterações entre diferentes arquiteturas. Outro fator importante na escolha de Java foi a arquitetura de componentes de *software* JavaBeans [SUN2000a]. A existência de um padrão nessa linha permite que uma grande quantidade de componentes já existentes possa ser integrada naturalmente às aplicações criadas com esta ferramenta.

Muitas idéias usadas neste trabalho e relacionadas à engenharia de *software* foram baseadas em ferramentas semelhantes de programação centralizada e que em certos casos são utilizadas também em programação distribuída. Entretanto, a programação distribuída nessas ferramentas sempre trata um lado da aplicação de cada vez, ou seja, o cliente e o servidor são desenvolvidos separadamente. Não é possível através de ferramentas como o JBuilder [BOR2000b], Microsoft Visual J++ [HOL98], Symantec Café [COR98] e IBM VisualAge [MAR99] se ter uma visão única da estrutura geral da aplicação, de seus objetos e relacionamentos. Com relação a essas ferramentas, a eliminação desta deficiência é o principal alvo da ferramenta proposta.

Outra deficiência atual a ser resolvida aqui se encontra nas ferramentas de programação paralela e distribuída citadas anteriormente (como o VPE e o CODE). Ao contrário das ferramentas de programação visual centralizada, o grupo de VPE e CODE

possui a capacidade de representar a estrutura e relacionamentos das aplicações, mas apresenta poucas facilidades em termos de desenvolvimento de *software*. Na sua grande maioria, não permitem a reutilização de componentes e nem ao menos utilizam uma linguagem de programação orientada a objeto. Essa dificuldade, quando se tratam de aplicações com grande complexidade, é um detalhe que influi consideravelmente no desenvolvimento das aplicações.

Em vista das características desses dois tipos de ferramentas, este trabalho procura apresentar uma ferramenta cujo alvo sejam as aplicações distribuídas, mas que seja dado também um suporte melhor à engenharia de *software*. As prioridades serão a facilidade de uso da ferramenta, portabilidade das aplicações e reutilização de código. Outro objetivo importante é a flexibilidade das aplicações. A ferramenta foi projetada e implementada com a finalidade de que represente um conjunto rico de aplicações e assim seja suficientemente flexível. Naturalmente, haverá pendências importantes para certos tipos de aplicações distribuídas e com isso será deixada para versões futuras a implementação de recursos importantes, como o suporte a aplicações tolerantes a falhas e agregação de um sistema de monitoração e visualização das aplicações.

Ao final do trabalho deseja-se ter como resultado uma ferramenta de desenvolvimento cuja principal contribuição seja a possibilidade de se identificar os relacionamentos entre os diferentes objetos de uma aplicação distribuída de forma clara e imediata por parte do usuário. Isso será feito através da programação visual que consegue tratar essa questão de maneira muito mais eficaz do que a programação textual. Além disso, com o conceito de componente, esta ferramenta objetivará aplicar noções importantes de engenharia de *software*, oferecendo a possibilidade de reutilização e encapsulamento do código dos programas.

1.4 Estrutura do texto

Reconhece-se que o texto dessa dissertação é longo. Entretanto, a sua extensão se justifica pela tentativa de apresentar o trabalho com alto grau de detalhe, a fim de que seus usuários e futuros desenvolvedores venham a encontrar neste texto a fonte principal para a solução da maior parte de suas dúvidas e problemas. Além disso, a dimensão do problema atacado, a preocupação com a apresentação (figuras, tabelas) e um bom número de exemplos contribuíram bastante para aumentar o tamanho do texto.

Toda ferramenta de programação é capaz de abrir grandes possibilidades de expansão e esta não é uma exceção (a lista de trabalhos futuros no final da conclusão é uma pequena amostra do que ainda pode ser feito). Por isso, espera-se que este trabalho seja o ponto de partida para uma série de projetos que certamente necessitarão de um bom suporte de documentação. Procurou-se justificar com cuidado as escolhas feitas a fim de que futuros projetos tenham uma base sólida para o seu prosseguimento.

Após esta introdução, o capítulo 2 discutirá os principais conceitos envolvidos ao longo deste trabalho, como programação visual e objetos distribuídos. A seguir, o capítulo 3 descreve as características da ferramenta proposta, seus requisitos e objetivos. O modelo de programação visual recebe destaque neste capítulo, seguido pelo capítulo 4 que apresentará os principais aspectos de implementação. Na seqüência, o capítulo 5 compara este trabalho com os outros tipos de soluções apresentados para os mesmos problemas que se pretendem resolver aqui. Por fim, o trabalho encerra com uma conclusão que resume as principais contribuições obtidas e aponta trabalhos futuros.

2 Ambientes de programação

Neste capítulo serão estudados os principais conceitos empregados no desenvolvimento do ambiente de programação descrito neste trabalho. Estes conceitos são utilizados em muitos tipos de ferramentas de programação referidas aqui: programação textual, programação visual, programação centralizada e programação distribuída. Cada uma dessas ferramentas possui suas vantagens e desvantagens, cuja análise sustenta o projeto de um novo ambiente que aproveite as melhores características de cada uma.

O estudo se iniciará pela definição de programação visual, a particularidade básica da ferramenta proposta. Seguirá com ambientes de programação, programação orientada a eventos, modelo de componentes de *software* e objetos distribuídos, conceitos também muito importantes neste trabalho. Ao final, tem-se uma rápida análise das ferramentas de desenvolvimento rápido de aplicações, destacando-se as de programação visual em Java para a plataforma Windows.

2.1 Programação visual

O estudo das linguagens de programação visual é relativamente recente. Os primeiros trabalhos desenvolvidos nesta área começaram com dois tipos de linguagens [BUR95a]: abordagens visuais a linguagens de programação tradicionais (tais como fluxogramas) e abordagens visuais diferentes da programação tradicional (programação por demonstração, por exemplo). Com o aprimoramento tecnológico dos dispositivos de entrada e saída (principalmente gráficos), as linguagens visuais foram levadas mais a sério em vista dos benefícios que elas poderiam trazer à programação de computadores. Se o aparecimento de dispositivos de exibição gráfica com preço e qualidade desejados tivesse ocorrido antes, certamente esta linha de pesquisa estaria muito mais desenvolvida. Ainda no tempo das telas baseadas em caracteres havia pessoas se esforçando para produzir ambientes visuais.

Segundo [GLI84], uma linguagem ou sistema de programação é classificado como visual, em oposição a textual, se uma ou ambas das seguintes condições seja(m) satisfeita(s):

- entidades gráficas de nível mais alto estão disponíveis ao usuário como átomos que ele pode manipular;
- elementos gráficos (que podem conter textos e números como componentes) formam uma parte integral (não meramente decorativa) da exibição gerada pelo sistema para usuários que estejam programando ou executando uma aplicação.

Resumindo, um programador de uma linguagem visual é capaz de expressar uma ação ou estado de uma aplicação através de gráficos. Embora essa não seja a única definição existente, pode-se aceitá-la como correta tendo em vista o caráter informal dessa definição. Uma definição ligeiramente diferente não alteraria o modo como o usuário enxerga um sistema visual, até porque os construtores das ferramentas não estão preocupados com uma definição exata de linguagem visual.

Um exemplar característico da programação visual é encontrado na ferramenta Prograph [PIC96]. Prograph é um ambiente visual de programação onde todas as ações dos programas são especificadas através de símbolos e manipulações visuais. Prograph é um ambiente destinado a aplicações de propósito geral, mas tem recebido extensões

para tratar problemas específicos como aplicações paralelas [COX98]. A FIGURA 2.1 mostra parte da implementação do algoritmo de classificação de dados *quicksort* através da linguagem visual de Prograph. Como se percebe, o programa é especificado somente através de símbolos pictóricos.

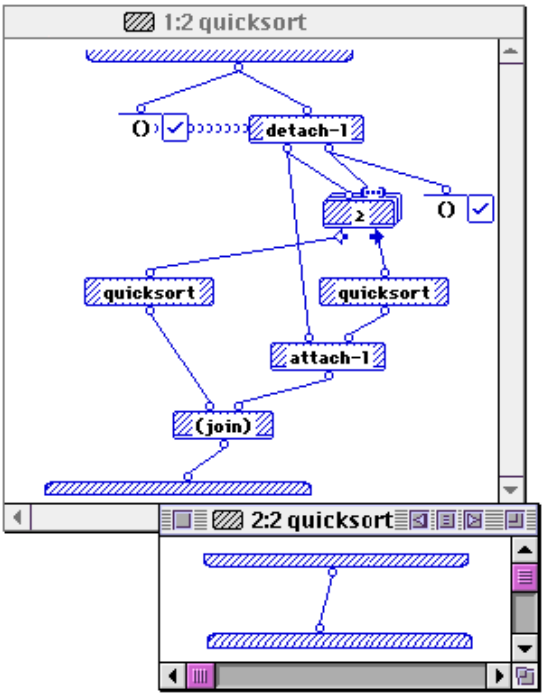


FIGURA 2.1 - Ambiente de programação visual Prograph.

Aproveitando muitas das características próprias de representações gráficas, como a facilidade de representação de um maior número de dimensões, pesquisadores de diversas partes do mundo procuram meios para transformar a tarefa de programação numa atividade o mais natural possível, onde a solução do problema esteja mais próxima do homem do que da máquina. Sob esse ponto de vista, percebe-se claramente que as representações abstratas providas pelas linguagens visuais são mais adequadas aos seres humanos do que aos computadores, ao contrário do que acontece com as linguagens textuais. As linguagens textuais foram projetadas inicialmente pela facilidade com que o computador tem em processá-las, pois, embora sejam capazes de representar praticamente tudo, fazem isso num nível mais próximo da maneira com a qual o computador e não o ser humano trabalha.

Atualmente, as linguagens visuais já têm suas qualidades reconhecidas. Entretanto, logo após o seu surgimento, as dificuldades da programação visual com o tratamento de programas de tamanho real levaram ao desencanto com este paradigma de programação. Muitos chegaram a acreditar em que ela era apenas um exercício acadêmico inadequado ao trabalho "real" [BUR95a]. Para superar parte desse problema, os pesquisadores começaram a desenvolver meios para usar a programação visual apenas em determinadas fases do desenvolvimento do *software* [BUR95a]:

- programação de interfaces gráficas;
- determinação gráfica de relacionamentos e comportamentos de estruturas de dados;
- combinação visual de unidades independentes para compor novos programas.

Como será visto em outra seção, a programação visual paralela e distribuída se enquadra nos dois últimos itens, onde os gráficos são usados para especificar relacionamentos entre as partes concorrentes do programa e combinar essas mesmas partes para formar uma aplicação completa.

As linguagens de programação visual empregam técnicas visuais para tentar possuir uma ou mais das seguintes características [BUR95]:

- **número menor de conceitos para programar**: por exemplo, em muitas linguagens de programação visual, o programador não precisa lidar com ponteiros, alocação de memória, declarações, escopo ou variáveis;
- **processo de programação concreto**: o programador pode ver, explorar e mudar valores específicos de dados ou ainda amostras de execuções;
- **exibição explícita de relacionamentos**: em determinados modelos de programação, é muito importante se ter uma visão geral dos relacionamentos e composições entre os objetos. Como exemplo, pode-se citar os diagramas de fluxo de dados e a programação paralela explícita, onde os relacionamentos são traduzidos pela comunicação e sincronização entre os processos. A ferramenta proposta neste trabalho se vale dessa particularidade da programação visual para alcançar um de seus principais objetivos que é a identificação dos relacionamentos existentes entre os diferentes objetos de uma aplicação;
- **retorno visual imediato**: as ações executadas pelo programador são refletidas na saída imediatamente. Se alguma alteração no código é feita, todo o resto do programa que for dependente desta alteração é atualizado automaticamente.

Observa-se que a maioria dessas características está relacionada com a facilidade de programação, onde o usuário dispõe de uma linguagem simples que abstraia as construções de mais baixo nível. A possibilidade de se ter uma visão geral do programa e de se exercer um controle mais elaborado sobre ele são outros fatores de impulso positivo na busca da facilidade de programação.

A linguagem de programação visual é a base dos ambientes de programação visual, mas a facilidade de uso não se limita somente a ela. Assim como na programação seqüencial, deseja-se que as ferramentas de desenvolvimento não sejam formadas apenas por editores e compiladores pouco integrados. Os novos ambientes de programação combinam numa mesma interface todos os recursos necessários, desde a programação até a depuração com a finalidade de aprimorar todo o processo de desenvolvimento. A integração dos módulos permite maior controle sobre todas as tarefas, pois a mesma interface com o usuário é utilizada continuamente.

2.1.1 Programação textual concorrente

A maior parte do desenvolvimento em sistemas distribuídos e paralelos é levada a termo através de ferramentas de programação textual. Antes de se analisar a programação visual concorrente, vale a pena conhecer os principais tipos de ferramentas de programação textual paralela e distribuída e suas peculiaridades mais importantes:

- **linguagens de programação concorrente**: são linguagens que suportam concorrência de forma nativa através de construções embutidas na própria linguagem. As mais conhecidas são SR [AND88], Linda [AHU86][CAR89][CAR89a], Orca [BAL88][BAL90][BAL91][BAL92] e CC++ [SIV94];

- **linguagens de programação seqüencial mais bibliotecas de programação**: as linguagens e compiladores utilizados são os mesmos da programação seqüencial e a concorrência e comunicação são implementadas através de bibliotecas de funções ou de classes. Exemplos: Java e suas bibliotecas de comunicação por *sockets* e concorrência com *threads*, C [KER78] ou FORTRAN [DIA71] com as bibliotecas PVM [BEG91] ou MPI [MPI94];
- **compiladores e preprocessadores paralelizadores**: estes sistemas tomam um programa codificado numa linguagem de programação seqüencial e descobrem as partes que podem ser paralelizadas ou distribuídas. Os elementos de código mais comuns envolvidos nesse processo são as estruturas de dados vetoriais e as instâncias de objetos. Exemplo: PGI [PGI2000], DPC++ [CAV93].

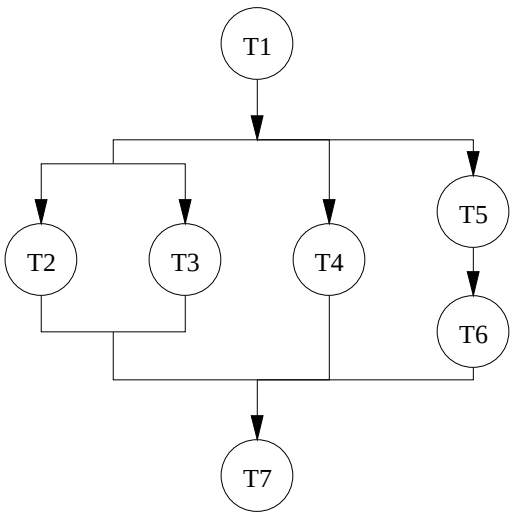
Cada um desses tipos de ferramenta implementa as soluções dos problemas concorrentes de uma maneira diferente, mas todos possuem as limitações da representação unidimensional dos programas imposta aos programas paralelos e distribuídos. Como mostra a seguinte seção, um arquivo texto formado por uma seqüência linear de caracteres não é a melhor escolha para se descrever tais programas.

2.1.2 Programação visual concorrente

Linguagens de programação concorrente totalmente baseadas em representações textuais não são a melhor escolha para se descrever a concorrência. As desvantagens principais vêm do fato de que a ordem seqüencial do texto esconde a estrutura multidimensional dos programas [WIR94b]. Assim, utilizada na programação centralizada há mais tempo, a programação visual ultimamente vem sendo pesquisada com respeito aos benefícios que ela pode oferecer à programação distribuída. Intuitivamente, é de se esperar que uma representação visual de um programa concorrente exiba a informação de maneira mais clara e organizada, possibilitando que a compreensão da estrutura geral do programa seja feita com menor esforço. Pode-se perceber isso através da análise de um programa concorrente qualquer (FIGURA 2.2), do qual se têm as duas representações: uma descrição textual baseada numa linguagem de programação concorrente (CC++) e sua descrição visual correspondente.

```
T1.run();
par
{
    par
    {
        T2.run();
        T3.run();
    }
    T4.run()
    {
        T5.run();
        T6.run();
    }
}
T7.run()
```

(a)

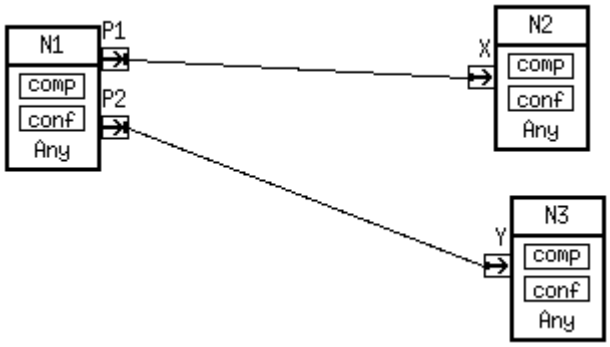


(b)

FIGURA 2.2 - Programa representado em CC++ (a) e graficamente (b).

Para os acostumados com a programação seqüencial, a representação textual sugere que a tarefa T2 seja executada antes da tarefa T3. A semântica da linguagem CC++ contudo não é essa. T2 e T3, por fazerem parte de um bloco *par*, são executadas concorrentemente. Esse erro de interpretação é mais difícil de acontecer na análise da representação visual, onde se percebe com muito maior facilidade que as tarefas T2 e T3 são executadas simultaneamente.

O exemplo acima demonstrou a capacidade maior que os gráficos têm de representar a estrutura das aplicações concorrentes. Outro tipo de análise pode ser feito quanto aos relacionamentos existentes entre os objetos das aplicações. Com o exemplo a seguir será fácil perceber a utilidade das representações visuais na expressão dos relacionamentos entre os diversos participantes de uma computação concorrente. A FIGURA 2.3 exibe a representação gráfica e textual na ferramenta VPE [NEW95] de um programa onde existem três processos se comunicando.



N1

```
int v = 4;

vpe_psend(&v,1,VPE_INT,P2); // envia v -> porta de saída P2
vpe_psend(&v,1,VPE_INT,P1); // envia v -> porta de saída P1
```

N2

```
int v, n;

vpe_precv(&v,1,VPE_INT,&n,X); // recebe v na porta de entrada X
printf("N2 recebeu o valor %i\n", v);
```

N3

```
int v, n;

vpe_precv(&v,1,VPE_INT,&n,Y); // recebe v na porta de entrada Y
printf("N3 recebeu o valor %i\n", v);
```

FIGURA 2.3 - Programa representado graficamente e textualmente em VPE.

Qualquer um pode perceber imediatamente qual é a representação mais apropriada à expressão de relacionamentos. Enquanto que o código textual necessita de uma análise mais cuidadosa em todos os aspectos, desde os identificadores das funções até seus parâmetros e variáveis, no grafo os relacionamentos entre os diferentes objetos podem ser visualizados diretamente. As setas indicando as trocas de mensagens servem perfeitamente como analogias aos canais de comunicação reais que fazem parte do conjunto da aplicação.

Com esses exemplos conclui-se que as linguagens de programação visual possuem a mesma capacidade de expressão das linguagens textuais, mas, por serem mais amigáveis aos olhos do programador, conseguem traduzir mais fielmente os seus desejos, tornando assim mais simples a tarefa de desenvolvimento. Resumindo, uma

exibição gráfica tem a vantagem de permitir ao programador que compreenda os aspectos de uma computação paralela com maior rapidez e precisão [ZER92].

Mas a programação não é o único benefício da visualização em programação concorrente. A visualização em aplicações concorrentes pode ser usada para dois propósitos adicionais [ZHA94][WIR94a]:

- **depuração**: a localização e correção dos erros dos programas são feitas de forma visual. A visualização é importante porque além das duas dimensões naturais da programação concorrente, a depuração acrescenta ainda a dimensão tempo. O grande número de objetos que devem ser observados simultaneamente numa sessão de depuração é outro motivo para que se encontre uma maneira mais confortável de se expressar o estado de uma computação, pois o controle textual de vários fluxos de execução já é por si só difícil de gerenciar;
- **sintonia de desempenho**: a compreensão geral do programa é condição valiosa para a obtenção de um bom desempenho. Por mostrar com maior clareza a estrutura geral e os relacionamentos do programa, a visualização permite que o programador o compreenda melhor.

Outra grande vantagem da utilização de elementos visuais na programação concorrente é a possibilidade de se ter uma mesma representação durante todas as fases do desenvolvimento do programa [CAI95][BRO94]. A mesma representação é usada tanto para especificar quanto para depurar e visualizar os programas. A integração destas três fases torna todo o processo mais simples e a ferramenta mais fácil de ser usada, pois o usuário poderá aproveitar sua experiência no uso de diferentes módulos.

Seguindo na busca pela facilidade de uso, um ambiente de programação visual, como qualquer outro ambiente de programação, deve oferecer facilidades ao usuário para libertá-lo dos níveis mais baixos de implementação. Considerando a programação concorrente, o programador não deve ser obrigado a conhecer a estrutura interna da comunicação, como por exemplo as chamadas de biblioteca utilizadas para enviar ou receber uma mensagem. Isto é válido principalmente para um ambiente de programação que procure oferecer portabilidade, pois a parte de mais baixo nível dependente de plataforma é gerenciada pela ferramenta. A programação visual é importante neste caso porque a representação gráfica dos elementos de um programa é feita num nível de abstração onde os detalhes de cada plataforma não são visíveis. Somente no último estágio da implementação a ferramenta se encarregará de traduzir esses elementos para o código da plataforma onde o programa irá executar.

Como visto até agora, as linguagens de programação visual são bastante adequadas à representação de programas concorrentes. Todavia, isto é verdade apenas para os elementos bidimensionais do programa: distribuição dos objetos, sincronização e outros relacionamentos. Para os elementos seqüenciais, como o código interno dos processos, a representação textual ainda é a melhor saída para a programação. Algumas ferramentas [WIR94][KAC96] continuam representando o código interno dos processos de forma visual, mas fazem isso apenas em construções de mais alto nível. Nessas ferramentas, o código interno de cada processo é subdividido em construções visuais representando, por exemplo, envio e recebimento de mensagens para que elas sejam gerenciadas visualmente com uma granulosidade menor. Quando isto não é feito, o que acontece é que os processos enviam e recebem mensagens somente no início ou no fim de momentos predefinidos, o que limita a flexibilidade da comunicação.

2.1.3 Modelo de Grafos

Um dos modelos mais utilizados na representação visual de programas concorrentes é o modelo de grafos. Há várias ferramentas de programação que o usam tanto para programar como para visualizar e depurar a aplicação. Além do que já foi apresentado a respeito das linguagens visuais, há ainda outras vantagens atribuídas a uma representação de programas concorrentes através de grafos [BRO94]:

- os grafos são uma representação mais natural para programas concorrentes do que textos lineares, porque o comportamento desses programas é multidimensional;
- uma linguagem baseada em grafo pode separar a programação em dois níveis, criando elementos seqüenciais e os compondo em um programa concorrente completo, facilitando o método de divisão e conquista;
- os grafos exibem diretamente a estrutura complexa que os programadores devem entender para obterem alto desempenho;
- a representação visual promove a exploração da localidade de dados, outra questão chave no desempenho desses tipos de programas;
- um modelo de grafo permite que a depuração e a avaliação do desempenho sejam feitos no mesmo ambiente de programação.

Como não poderia deixar de ser, o modelo de grafos para programas concorrentes apresenta suas desvantagens. A representação de um programa paralelo ou distribuído através de grafos não é adequada para programas de grande porte, pois não é uma tarefa trivial se obter um programa legível de um grafo arbitrariamente desenhado [CAI95]. Este problema, uma das conseqüências da escalabilidade deficiente das linguagens visuais, é verificado em [CAI95] e a solução apontada são os mapas de concorrência, outro tipo de representação visual para programas concorrentes. Entretanto, também não se pode afirmar que esta solução resolve totalmente o problema da escalabilidade.

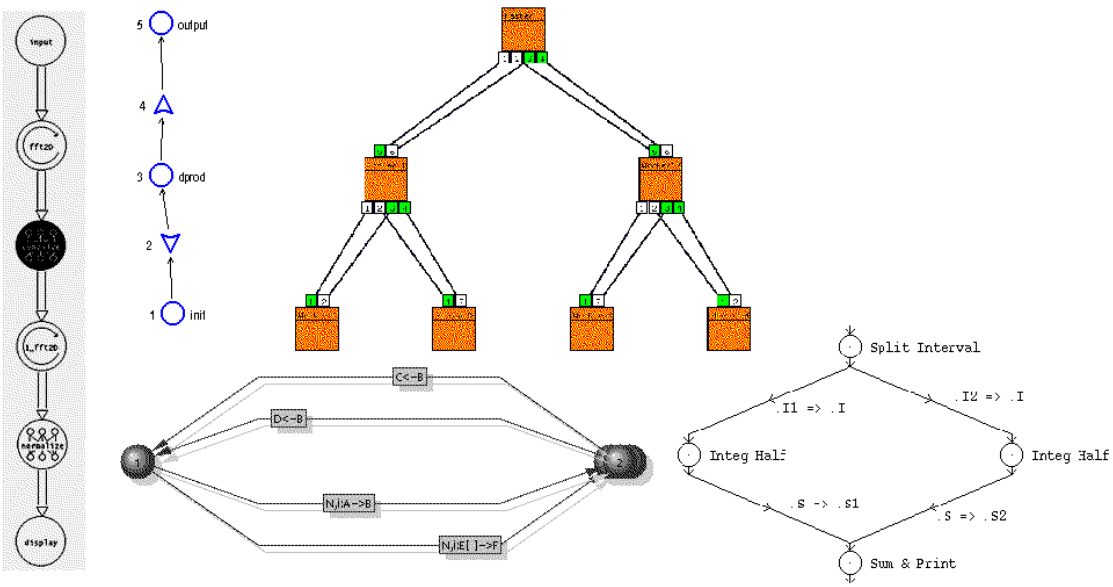


FIGURA 2.4 - Grafos em diferentes ferramentas de programação visual paralela.

Há vários modelos de grafos largamente utilizados para construir programas, como Redes de Petri, Grafos de Dependência de Programas, Grafos de Processos e notações baseadas em formulários [ZHA94]. Muitos sistemas usam inclusive mais do

que um modelo de grafo simultaneamente, mas a maioria das ferramentas existentes atualmente emprega os modelos mais conhecidos de grafos, como mostra a FIGURA 2.4.

2.1.4 Questões em Programação Visual Paralela e Distribuída

Uma das grandes questões envolvidas na pesquisa das linguagens de programação visual em geral, não somente programação visual paralela e distribuída, é a questão da maior eficácia e eficiência do uso destas linguagens em relação às linguagens tradicionais. Embora a maioria dos pesquisadores concorde sobre a validade das linguagens visuais na programação, há opiniões em contrário.

É fácil de se implementar uma linguagem que seja visualmente agradável, mas é muito mais difícil se conseguir tal nível de satisfação em relação à capacidade da linguagem de expressar as necessidades do problema computacional. O que se percebe comumente são linguagens visuais que de fato conseguem expressar um programa, mas que após uma análise mais profunda são preteridas em relação às tradicionais linguagens textuais.

As dificuldades presentes no projeto das linguagens visuais devem-se muito também aos tipos de aplicação que elas se propõem a representar. Não são todos os tipos de problemas e soluções que se adaptam bem a uma descrição visual. Em alguns casos, como por exemplo na descrição de fórmulas matemáticas, estudos demonstraram que pode ser muito mais rápido e simples se descrever uma solução textual do que se fazer o mesmo através de elementos gráficos.

Entretanto, a programação paralela e distribuída é um campo de estudo que se adapta bem às soluções apresentadas pelas linguagens de programação visual. É consenso entre muitos autores que o grafo é uma representação natural para programas concorrentes [WIR94a]. Apesar do esforço produzido na pesquisa da programação visual para aplicações paralelas e distribuídas, a aceitação deste tipo de ferramenta ainda é baixa entre os usuários dos ambientes concorrentes. Há várias razões para esse comportamento, mas a principal é a ligação com os ambientes de programação textual que há anos vêm sendo utilizados com resultados razoavelmente eficazes. Os programadores passaram anos trabalhando de uma maneira que lhe vem trazendo bons resultados, o que torna a mudança de atitude um novo desafio. Se a maneira com que as coisas estão sendo realizadas continua resolvendo os problemas satisfatoriamente, o novo paradigma teria que ser muito melhor para justificar um investimento numa nova metodologia. Outro fator de influência é a possibilidade de se ter que reescrever parte do código já existente. Apesar de que muitas ferramentas sejam capazes de aproveitar código legado, isto dificilmente acontece de forma totalmente transparente e no final das contas acaba sempre exigindo um esforço extra de portabilidade.

2.2 Ambientes de programação paralela e distribuída

A visualização é a base de muitos ambientes de programação, mas não é o único recurso desses ambientes. Ela é um meio para que os ambientes integrados de desenvolvimento ofereçam ao usuário uma única interface para todas as fases da criação do *software*, facilitando o desenvolvimento e a busca de produtividade e eficiência. A larga utilização dos ambientes integrados de desenvolvimento na programação seqüencial tem mostrado que eles são extremamente válidos e realmente úteis na busca desses objetivos. Cada vez menos é considerada a possibilidade de se usar, por exemplo, um compilador de linha de comando e um depurador externo que não foram projetados

para trabalharem de maneira integrada. Em geral, quando são usados com esse objetivo, não costumam apresentar os resultados esperados.

Por ser uma área de conhecimento relativamente recente, a programação paralela e distribuída ainda carece de ambientes bem estabelecidos, o que em muitos casos leva os programadores ainda a usarem ferramentas mais antigas e a questionarem o uso das novas. Além da inexperiência, outros fatores de ordem técnica e pessoal contribuem para que as novas ferramentas sejam preteridas em relação às outras. A resistência em aderir a novas tecnologias, o esforço adicional necessário para se aprender a usar uma nova metodologia e o compromisso com o investimento já feito são os principais fatores que dificultam a adoção de novas ferramentas.

Apesar e por causa dessa resistência, muito esforço e pesquisa estão sendo feitos atualmente para aumentar a qualidade das ferramentas utilizadas nos ambientes paralelos e distribuídos. Muitos seguem a tendência existente na programação de ambientes centralizados e investem na programação visual. Os ambientes de programação visual se caracterizam por apresentar uma linguagem visual como interface de programação, mas, além disso, devem apresentar uma série de requisitos, como facilidade de uso, eficiência e portabilidade para que sejam tecnologicamente bem classificados. Considerá-los ambientes completos de desenvolvimento para aplicações paralelas e distribuídas ainda exige que eles tenham mecanismos de visualização e depuração para avaliar o desempenho e a correção dos programas.

Embora a maioria dos pesquisadores considere a programação visual importante para o desenvolvimento de aplicações paralelas, há resistências à aceitação completa dessa idéia. Alguns ainda preferem continuar usando programação textual e aproveitar a experiência já adquirida nesse ambiente. Portanto, não é possível se afirmar categoricamente que as linguagens visuais serão úteis a todos os tipos de usuários para o desenvolvimento de qualquer tipo de aplicação. Embora sejam um grande avanço de engenharia de *software*, as ferramentas visuais devem ser vistas como um complemento e não como substituição aos modelos de programação tradicionais.

2.2.1 Requisitos de uma ferramenta de programação

É difícil ou talvez impossível projetar e construir uma ferramenta de programação ideal. Entretanto, é pelo menos desejado que as ferramentas busquem se aproximar do padrão ideal e que sejam realmente úteis aos seus usuários, mesmo que a satisfação não seja total. A busca dessa satisfação passa pelo atendimento de um certo número de requisitos que compõem uma situação ideal. A nova dificuldade que surge então é a definição de tais requisitos, pois a análise das ferramentas muitas vezes se torna uma tarefa subjetiva, que envolve o relacionamento homem-máquina e depende da opinião de vários tipos de usuários.

Muitos dos requisitos de uma ferramenta de programação paralela são os mesmos de uma ferramenta qualquer, como por exemplo facilidade de uso, eficiência e reutilização de código. Contudo, devido à maior complexidade da programação concorrente, a satisfação desses requisitos se torna mais crítica neste contexto. Sendo assim, os principais requisitos desses ambientes de desenvolvimento são [MAL99]:

- **facilidade de uso**: esta questão é difícil de ser avaliada, pois tem cunho subjetivo e assim varia entre diferentes classes de usuários. Algumas ferramentas são apropriadas a usuários mais experientes, enquanto outras se destinam a usuários novatos. É difícil adequar uma ferramenta aos dois tipos de

usuários [SZA94]. Os pontos da ferramenta onde a facilidade de uso possui maior impacto são o ambiente de programação e a interface com o usuário. Se bem projetados, eles podem ajudar muito o usuário a cometer menos erros e a desenvolver o seu programa de forma mais intuitiva;

- **eficiência**: a ênfase na facilidade de uso faz com que a ferramenta imponha ao usuário abstrações que se traduzem por camadas extras de *software* na implementação final, provocando perda de desempenho. Embora o desempenho seja o objetivo principal apenas da programação paralela, um *overhead* excessivo produzido pela ferramenta pode inviabilizar o seu uso também em aplicações distribuídas;
- **portabilidade**: em sistemas distribuídos é comum que estejam interconectadas diferentes plataformas de *hardware* e de *software*. É interessante portanto que as aplicações não estejam atreladas a uma plataforma específica, o que possibilita um compartilhamento maior dos recursos disponíveis;
- **flexibilidade**: a flexibilidade está relacionada com os tipos de aplicações que podem ser criadas com a ferramenta. Em sistemas distribuídos há diferentes tipos de interações entre os objetos que devem ser modeladas por uma ferramenta que deseje oferecer flexibilidade aos seus usuários. Entretanto, deve haver um compromisso com a facilidade de uso, pois quanto mais completa a ferramenta, geralmente mais complicado é o seu uso;
- **rapidez de desenvolvimento**: tempo é um fator crítico no desenvolvimento de *software*. As ferramentas procuram satisfazer este requisito buscando principalmente facilidade de uso, tirando do usuário a responsabilidade pela execução de muitas tarefas que podem ser automatizadas e diminuindo assim o tempo total de desenvolvimento;
- **reutilização de código**: a reutilização de código envolve o aproveitamento de programas antigos nas novas ferramentas e de módulos entre as novas aplicações. O primeiro caso é importante para a aceitação de uma nova ferramenta, pois muitos usuários não podem assumir a perda do investimento já feito com outras tecnologias. O aproveitamento dos módulos entre aplicações feitas na mesma ferramenta contribui para o desenvolvimento de bibliotecas de *software* e aumenta a produtividade e a qualidade das aplicações;
- **abstração da programação**: o oferecimento ao usuário de níveis de abstração mais altos evita que ele seja obrigado a lidar diretamente com detalhes de baixo nível das aplicações, contribuindo assim para a diminuição da quantidade de erros. Representações abstratas são também importantes para aproximar o usuário da solução do problema e não da sua implementação, que em casos como sistemas distribuídos costuma ser complexa;
- **compatibilidade**: para diminuir o tempo de aprendizado da ferramenta, ela deve aproveitar não somente tecnologias mas também culturas de programação. Uma ferramenta que introduza uma cultura de desenvolvimento totalmente diferente sofrerá maior resistência dos usuários experientes e mais acostumados a outras técnicas de produção. Esta é uma das razões pelas quais as ferramentas de programação visual combinam representações visuais com textuais na especificação dos programas, as quais são plenamente conhecidas e empregadas;

- **representação da concorrência**: o principal motivo para se introduzir a programação visual no desenvolvimento de aplicações paralelas e distribuídas é a maior facilidade de expressão da concorrência que os gráficos possuem em relação às ferramentas textuais. Uma forma adequada de representar a concorrência é fundamental para o sucesso da ferramenta.

Certamente outros requisitos podem ser citados, mas partindo desses pontos o desenvolvedor de uma ferramenta pode chegar a um resultado capaz de resolver os principais problemas do programador de aplicações paralelas e distribuídas. Com adaptações desses requisitos aos tipos de aplicações desejados, a ferramenta pode ser ainda mais útil aos seus usuários.

2.2.2 Ferramentas atuais de programação visual paralela e distribuída

As ferramentas atuais de programação visual paralela e distribuída cumprem os seus objetivos ao representarem visualmente aplicações concorrentes, mas ignoram outras questões também importantes para esses tipos de aplicações. Em primeiro lugar, as aplicações distribuídas recebem pouca atenção, já que a maioria das ferramentas gera código para as máquinas virtuais paralelas PVM e MPI que não são usadas na programação distribuída de sistemas abertos. A reutilização de código e de componentes é precária, devido principalmente à falta do conceito de objeto e de padrões preestabelecidos em tais ferramentas (maiores detalhes sobre as ferramentas de programação visual paralela e distribuída podem ser encontrados em [MAL99]).

Embora faltem estudos mais apurados sobre os reais benefícios da programação visual paralela e distribuída, o constante surgimento de novas ferramentas desse tipo mostra a crença de muitos pesquisadores no seu valor. Ao mesmo tempo, entretanto, sugere que o que existe atualmente apresenta deficiências que somente novas ferramentas podem suprir. Foi baseando-se nisso que a ferramenta proposta neste trabalho foi construída. Procurou-se assim aproveitar as vantagens dos ambientes atuais e resolver os principais problemas ainda não tratados. Uma comparação detalhada entre os aspectos técnicos das ferramentas será feita no capítulo 5, dedicado especialmente aos trabalhos relacionados.

2.3 Programação orientada a eventos

Uma parte importante do ambiente de desenvolvimento aqui apresentado exige a compreensão do modelo de programação orientada a eventos. Como será visto na parte de descrição e implementação da ferramenta, o modelo de eventos é usado na programação das aplicações e é peça chave no desenvolvimento de bons programas. Portanto, antes de desenvolver qualquer aplicação, o programador deve estar familiarizado com o paradigma de programação orientada a eventos, o que torna necessária uma introdução neste espaço a respeito desse assunto.

O elemento central desse paradigma de programação é o evento. O conceito de evento pode variar ligeiramente conforme o contexto, mas a definição básica é a de que um evento é uma notificação de alguma mudança de estado num objeto qualquer [KRI95], uma ocorrência específica de uma ação durante uma computação [PAN93] ou ainda qualquer influência externa. Reunindo as três idéias, um evento é o resultado da mudança de estado num objeto e que pode também influenciar o comportamento de outros objetos.

Programação orientada a eventos é portanto aquela onde os objetos da aplicação são tratados em função dos eventos que geram e esperam receber. Este paradigma é muito bem conhecido entre os desenvolvedores de interfaces gráficas e já se tornou um padrão para esta área. As interfaces gráficas mais conhecidas foram concebidas segundo esta filosofia, como o Microsoft Windows, X Window [QUE90], XView [HEL90], Motif [HEL91], Java AWT [SUN2000e] e a sua evolução Java Swing [SUN2000e]. Muitos ambientes de desenvolvimento para o ambiente Windows também utilizam esse paradigma como ferramenta principal de desenvolvimento de interfaces gráficas, como o Borland Delphi [BOR2000], Borland C++ Builder [BOR2000a], Borland JBuilder [BOR2000b] e Microsoft Visual Basic [PET98].

O paradigma de programação orientada a eventos surgiu da necessidade de se tratar ações geradas a partir de diversas fontes. Antes das interfaces gráficas, os programas costumavam realizar a entrada de dados apenas através do teclado e isso permitia aos programas executar comandos do tipo "ler(teclado)", pois essa era a única possibilidade de entrada de dados. Entretanto, com o surgimento de novos dispositivos de entrada como o *mouse*, não se podia mais conhecer previamente qual ação deveria ser executada. E também não era possível simplesmente se escolher entre tentar ler o teclado ou o clique do *mouse*. Por esse motivo surgiu a programação orientada a eventos. Neste modelo, ao invés de o programa obrigatoriamente ter que sincronizar com eventos externos e assim esperar que algo aconteça, ele é notificado de forma assíncrona a respeito da ocorrência de ações que lhe interessam. Esta é a principal diferença para a programação tradicional (*polling*), onde todas as diferentes fontes de eventos devem ser consultadas sempre que necessário. No exemplo citado acima, ao invés de ficar bloqueado numa chamada de leitura de teclado, o programa executa normalmente suas atividades sem se preocupar com a entrada e saída. Quando uma tecla é pressionada, o programa é notificado do evento e somente então a ação correspondente é tomada.

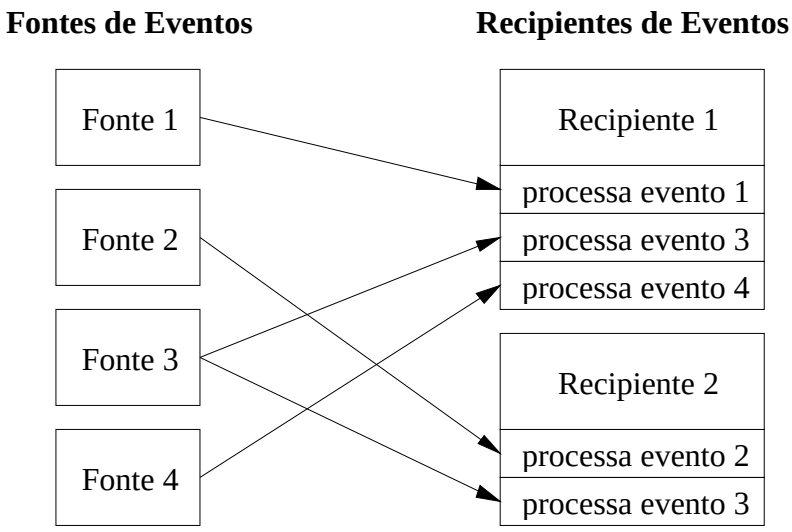


FIGURA 2.5 - Programação orientada a eventos.

A FIGURA 2.5 exibe a arquitetura básica da programação orientada a eventos. Nela, de um lado estão os objetos que geram eventos e de outro aqueles que consomem eventos. Naturalmente, a divisão não é exclusiva e permite que um objeto seja tanto produtor quanto consumidor de eventos. A mesma figura mostra também que é possível

que a uma mesma fonte de eventos possam estar associados diversos consumidores de eventos, configurando uma difusão de eventos (no exemplo, a fonte 3).

Quanto à implementação, há duas maneiras básicas de se construir um sistema baseado em eventos. A primeira (FIGURA 2.6), baseada no sistema de notificação, possui um agente notificador que recebe os eventos e chama rotinas registradas pela aplicação para tratar esses eventos. Em nível de aplicação, não é necessário que se lide diretamente com filas de eventos. A segunda abordagem (FIGURA 2.7), ao contrário da primeira, oferece à aplicação uma fila de eventos que deve ser lida e processada à medida que os eventos vão chegando. A aplicação fica num laço contínuo retirando eventos da fila e os processando à medida que eles chegam.

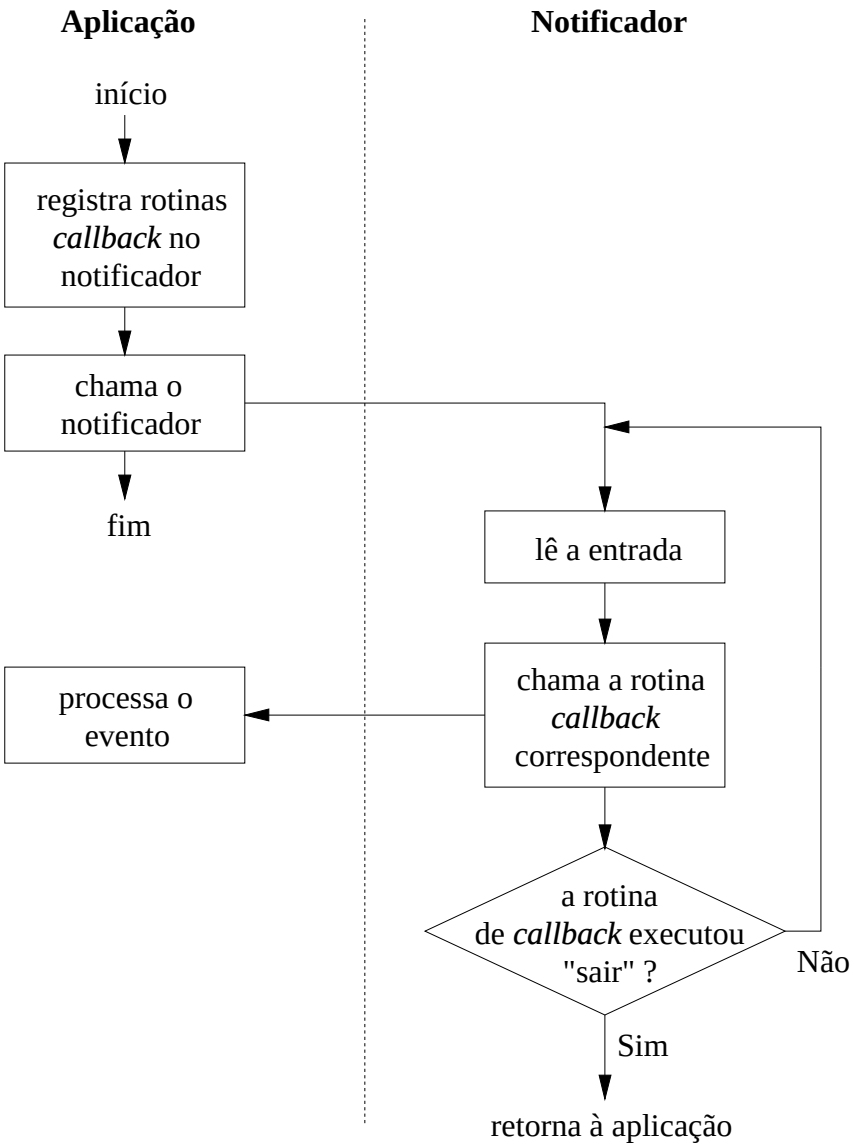


FIGURA 2.6 - Implementações de eventos utilizando notificador e rotinas *callback* [HEL90].

Nas duas opções deve ser construído um núcleo de gerenciamento de eventos que analisa as diversas fontes de eventos, buscando os eventos e chamando as rotinas correspondentes ou os colocando nas filas dos objetos consumidores. A diferença entre os dois é que no segundo caso o gerenciamento da fila de eventos deve ser

expressamente codificado na aplicação do usuário. No primeiro, o usuário deve apenas informar quais as rotinas a serem invocadas caso algum evento ocorra.

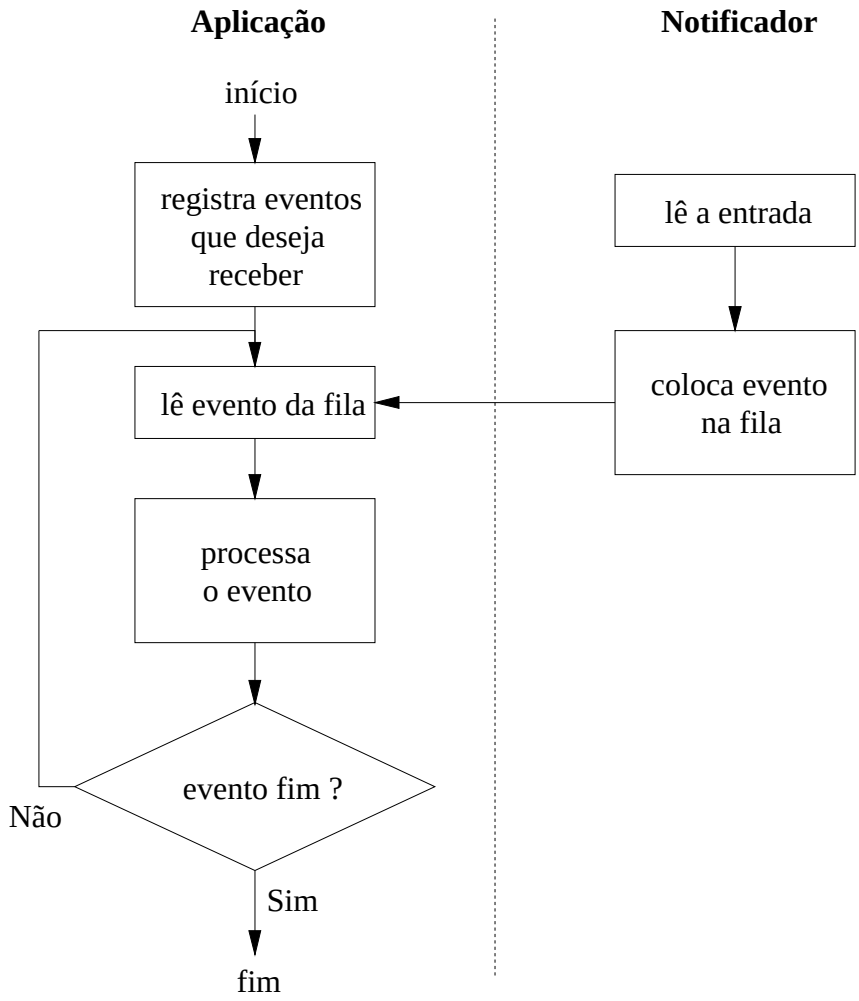


FIGURA 2.7 - Programação orientada a eventos usando laços de eventos.

Como era de se esperar, nem todos os tipos de aplicações se adaptam bem ao modelo de eventos. A própria programação paralela, que busca alto desempenho através de atividades longas que consomem uma grande quantidade de tempo não se encaixa bem neste modelo. No modelo de eventos, o tempo de processamento do evento deve ser mínimo a fim de que o período de espera dos eventos subseqüentes na fila não bloqueie demasiadamente o andamento normal da aplicação.

Outro problema enfrentado pela programação orientada a eventos é a dificuldade que programadores tradicionais encontram ao usar esta abordagem pela primeira vez. É um paradigma de programação com diferenças importantes que exigem um certo tempo de aprendizado e experiência para que bons resultados sejam alcançados. Uma disciplina de programação tradicional aplicada a um ambiente baseado em eventos traz ineficiência e cria implementações difíceis de serem mantidas.

2.3.1 Eventos em programação distribuída

A estruturação das aplicações em termos de eventos gerados externamente como cliques de botões e seleção de itens de menus é uma maneira natural de se tratar a

implementação de interfaces gráficas. O seu uso na programação geral, entretanto, é uma prática relativamente nova [CUG98]. Um exemplo fora das interfaces gráficas onde ela pode ser encontrada são os protocolos de enlace apresentados por Tanenbaum [TAN96], onde os estados dos protocolos são alterados em função dos eventos gerados pelas entidades remotas.

O principal motivo que leva ao uso da programação com eventos no tratamento de interfaces gráficas - as várias fontes de eventos - é o mesmo que porta para a programação distribuída esta técnica. O andamento de uma aplicação distribuída na maioria das vezes está condicionado à ocorrência de eventos em diferentes entidades remotas, podendo eles serem desde a presença de dados novos nas portas de comunicação até ações decretadas remotamente como o encerramento de um processo ou de uma conexão.

A programação tradicional por meio de *polling*, ou seja, verificando continuamente se há dados novos nas portas de comunicação ou se o estado dos objetos remotos foi alterado, pode tratar perfeitamente desse problema. Entretanto, a compreensão e implementação da solução são muito mais simples quando modeladas à luz do paradigma de eventos. A percepção de tudo isso tem crescido com o passar do tempo e atualmente os eventos distribuídos ou concorrentes existem em muitos ambientes: eventos distribuídos em Java utilizando RMI [SUN2000], ambiente Triveni [COL98][COL98a], ambiente JEDI (*Java Event-based Distributed Infrastructure*) [CUG98], CORBA Services [OMG98] e outros trabalhos que relacionam concorrência e eventos, como os estudos feitos em [STA95] e [PAN93].

As vantagens principais do paradigma de eventos em relação às técnicas tradicionais no contexto da programação distribuída são as seguintes:

- **concorrência**: por exemplo, quando um processo ou *thread* estiver pronto para receber uma mensagem, ele antes recebe o evento correspondente a essa mensagem e somente depois executa a chamada *receive*. Isso evita que o receptor fique bloqueado desnecessariamente na espera pela mensagem, permitindo que execute outras tarefas durante este período;
- **modularidade**: componentes descritos em função de eventos são facilmente modularizáveis, pois o conjunto de eventos pode ser usado como interface de comunicação entre componentes diferentes;
- **transmissor e receptor anônimos**: para receber um evento não é necessário conhecer o identificador do produtor do evento, assim como a produção de um evento não exige o conhecimento do receptor [CUG98]. A distribuição dos eventos aos seus destinos corretos é feita pelo sistema de controle de eventos que executa num nível mais abaixo;
- ***multicast***: implementado de forma trivial (vários receptores interessados no mesmo evento);
- **facilidade de desenvolvimento**: um mecanismo de invocação baseado neste paradigma encoraja o fraco acoplamento entre objetos que suportam alto grau de encapsulamento, simplificando o desenvolvimento de aplicações grandes e complexas [STA95];
- **reconfiguração dinâmica**: uma arquitetura baseada em eventos permite com mais facilidade a reconfiguração dinâmica do sistema, de modo que

componentes podem ser retirados e adicionados livremente por causa do fraco acoplamento do sistema [CUG98].

Como não poderia deixar de ser diferente, há também algumas desvantagens quando se opta pelo uso de eventos em sistemas distribuídos:

- **tempo de processamento**: um evento deve ter tempo de processamento mínimo para que os eventos seguintes não fiquem por muito tempo na espera. Isto pode se tornar um problema quando as atividades são necessariamente longas. Pode-se tentar diminuir a granulosidade, mas isto causa ineficiência;
- **depuração**: torna-se mais difícil, justamente pela característica assíncrona da geração dos eventos, onde a seqüência não pode ser prevista em tempo de desenvolvimento [PAT2000].

Programação com eventos distribuídos é uma tecnologia recente e a quantidade de aplicações desenvolvidas segundo ela ainda é pequena. Entretanto, se comparados com os seus benefícios, vê-se que os problemas são de relevância menor. Isto faz com que novas tecnologias surjam e outras sejam adaptadas com extensões para suportar esse paradigma e assim aumentar as possibilidades de escolha dos programadores.

2.3.2 Eventos em Java

Tendo em vista o contexto deste trabalho, é importante se estudar a implementação do modelo de eventos de Java. Este modelo é chamado modelo de delegação [FAR98] e se baseia na chamada de método simples. Neste modelo, os objetos que se interessam por um evento se registram no objeto que o produz e este, quando deseja disparar o evento, simplesmente executa a chamada a um método nos objetos que fizeram o registro.

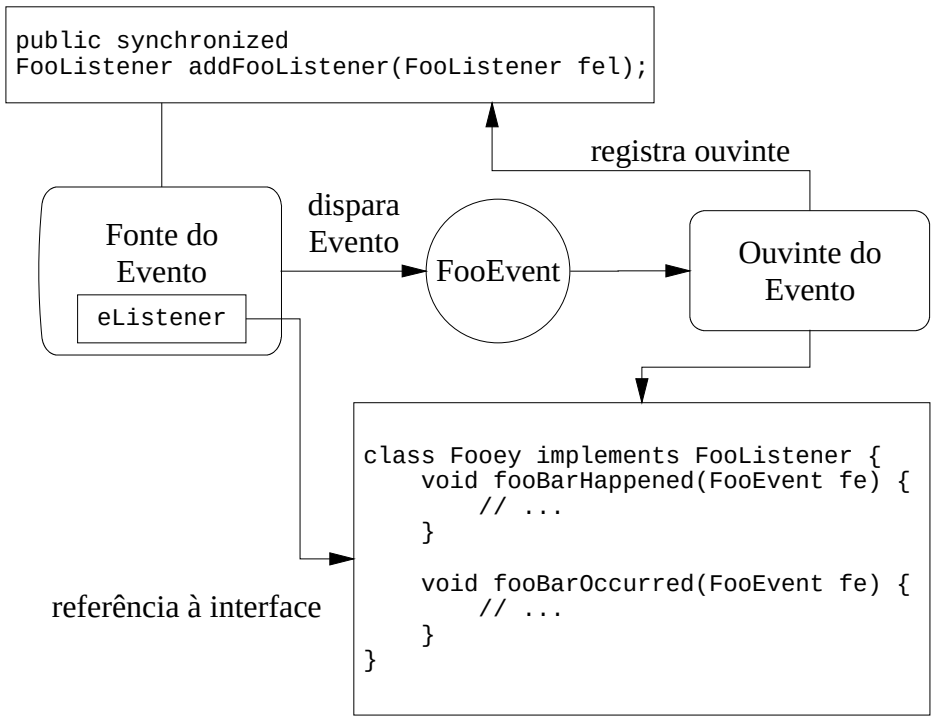


FIGURA 2.8 - Modelo de eventos em Java [SUN2000e].

A FIGURA 2.8 exibe uma visão geral do modelo de eventos em Java. A fonte do evento possui uma referência ao objeto que consome o evento (`eListener`), obtida através do cadastro que o consumidor fez com o método especial no objeto que gera o evento (método `addFooListener`). O objeto consumidor do evento deve implementar a interface correspondente (`FooListener`), pois o objeto que produz o evento irá executar métodos dessa interface quando algum evento for gerado.

Este modelo de eventos em Java é utilizado predominantemente na construção de interfaces gráficas com as bibliotecas AWT e Swing, mas qualquer outro tipo de aplicação pode ser programado com eventos em Java. Inclusive, está ainda em fase de definição a estrutura própria de eventos distribuídos de Java para a programação tanto de interfaces gráficas distribuídas como de outras aplicações em rede. A versão inicial da especificação está disponível em [SUN2000], onde podem ser conhecidos os detalhes da implementação dessa estrutura sobre RMI.

2.4 Modelo de componentes de software

Durante muito tempo os programadores de computadores implementavam um novo sistema totalmente a partir do zero. Qualquer nova aplicação tinha todas as suas funcionalidades implementadas e testadas independentemente de outros trabalhos que já tivessem sido feitos. Com o desenvolvimento da engenharia de *software* e o amadurecimento das tecnologias de orientação a objeto, começou a se perceber que na grande maioria das vezes as implementações das aplicações possuíam entre si vários pontos em comum. A partir de então surgiu a consciência de se padronizar módulos de *software* que pudessem ser compostos e aproveitados por implementações variadas.

Os primeiros resultados em grande escala da aplicação desses princípios foram as bibliotecas de rotinas ou de classes (para domínios gerais ou específicos) [SCH96a]. A partir de então, já não era mais necessário reimplementar toda a aplicação. Muito pelo contrário, era possível e até aconselhável usar módulos básicos já implementados, razoavelmente testados e documentados. Essa prática se tornou tão comum que hoje em dia é impossível se conceber uma aplicação de porte razoável sem o apoio de trabalho previamente desenvolvido. Com um conjunto rico de classes, as linguagens de programação orientada a objeto, como Java, são exemplos fortes da importância que as bibliotecas de código têm no cenário atual do desenvolvimento de *software*.

Com o passar do tempo, os conceitos e as tecnologias básicas foram se aprimorando e chegaram ao que hoje se denomina Engenharia de *Software* Baseada em Componentes (CBSE - *Component Based Software Engineering*). A CBSE unifica conceitos de um número de domínios de *software*, tais como programação orientada a objeto, arquitetura de *software* e computação distribuída [KOZ98] para chegar a um modelo de programação com componentes. O componente é o elemento central deste modelo e sua definição possui várias versões [BRO98]:

- um componente é uma parte não trivial, quase independente e substituível de um sistema que preenche uma função clara no contexto de uma arquitetura bem definida. Um componente segue e oferece a realização física de um conjunto de interfaces;
- um componente de *software* em tempo de execução é um pacote dinamicamente atável de um ou mais programas gerenciados como uma unidade e contatados através de interfaces de documentos que podem ser descobertas em tempo de execução;

- um componente de *software* é uma unidade de composição com interfaces especificadas contratualmente e que possui somente dependências de contexto explícito. Um componente de *software* pode ser desenvolvido independentemente e está sujeito à composição por terceiros;
- um componente de negócios representa a implementação de *software* de um conceito de negócio "autônomo" ou processo de negócio. Ele consiste dos artefatos de *software* necessários para expressar, implementar e desenvolver o conceito como um elemento reutilizável de um sistema de negócios mais amplo.

Combinando as definições, um componente de *software* é um elemento de natureza de alta granulosidade, com interfaces bem definidas e associado a um contexto com dependências explícitas, o que restringe a comunicação entre diferentes componentes somente à interface. A natureza clara das interfaces permite também que um componente possa se combinar com outros em tempo de projeto e de execução.

O desenvolvimento de *software* baseado em componentes é fundado no princípio da divisão e conquista: sistemas maiores precisam ser divididos e resolvidos por subsistemas individuais. O sucesso depende das unidades de *software* que foram desenvolvidas em outras situações, o que requer que se repense e atualize as soluções tradicionais de manutenção de *software* [VOA98]. É muito inspirado na implementação de *hardware*, onde na quase totalidade das vezes um novo módulo é construído a partir da composição de elementos já prontos. Há uma "biblioteca" de componentes, com suas interfaces bem especificadas e documentadas, pronta para ser usada: circuitos integrados, placas, portas lógicas e um sem fim de outros componentes eletrônicos.

Várias tendências atuais encorajam o uso de componentes reutilizáveis no desenvolvimento de aplicações. As razões econômicas e de competição entre os produtores de *software* exigem diminuição no orçamento de desenvolvimento e manutenção, assim como se impõem as necessidades de rápida prototipação e flexibilidade criadas a partir dos requisitos flutuantes causados pelo próprio domínio da aplicação [STA94]. Além dos fatores econômicos, tecnicamente existem vários argumentos motivadores para a utilização de componentes [KLA99]:

- reutilização de projeto e implementação;
- aumento de confiabilidade, pois em geral o código já foi bastante testado;
- ganho de tempo e diminuição do custo de manutenção (desenvolvimento menor de código);
- encapsulamento;
- possibilita soluções padronizadas de forma mais rápida;
- permite incluir novas funcionalidades a qualquer tempo ao adicionar novos componentes a uma solução já existente;
- troca de componentes por outros de versões mais recentes (exemplo: *softwares* que trabalham com legislações de um país, sujeitas a constantes alterações).

Apesar de ter muitas características relevantes ao desenvolvimento de programas, muito mais importante do que o tempo reduzido de projeto e implementação é a confiabilidade maior que um componente já pronto oferece. Na medida em que os componentes são utilizados por muitas pessoas, eles passam por um número maior de

testes, ganhando credibilidade em função da sua experiência no mercado. Mas o mais interessante nisto é que esse aspecto é paradoxal, pois se pode perceber também que, assim como outras questões, a confiabilidade é uma das desvantagens da engenharia de *software* baseada em componentes:

- falta de confiabilidade: embora os desenvolvedores de código afirmem a qualidade do *software*, nunca se pode confiar totalmente em código fornecido por terceiros. A característica de “caixa preta” dos componentes tende a iludir seus usuários e a passar a impressão de que seu funcionamento está livre de erros;
- necessidade de um tempo extra para leitura da documentação e aprendizado do uso do componente;
- necessidade de testar todos os componentes novamente, pois o novo contexto pode mudar completamente o comportamento [WEY98]. A falta de novos testes pode causar resultados desastrosos;
- possibilidade de o desenvolvedor do componente cessar o suporte à manutenção do componente [VOA98];
- cavalos de Tróia [VOA98];
- atualizações no componente para adição de novas funcionalidades ou correção de *bugs* podem torná-lo incompatível com sistemas anteriormente suportados [VOA98].

As vantagens, entretanto, são bem maiores. A prova está na existência de um sem número de ferramentas de desenvolvimento baseadas em componentes de *software* e no surgimento a cada dia de novas arquiteturas de componentes. Um exemplo importante é a arquitetura de componentes da linguagem Java, chamada JavaBeans, descrita brevemente a seguir.

2.4.1 JavaBeans

Java possui muitas características voltadas para a reutilização de componentes de *software*: comentários, interfaces, classes, exceções, pacotes, constantes nomeadas, *garbage collection*, *multithreading*, herança, sobrecarga e suporte à reutilização do conjunto atual de APIs [TRA97]. Além de tudo isso, foi criada a arquitetura de componentes JavaBeans, onde através de sua interface de programação é possível construir componentes reutilizáveis e independentes de plataforma. Com o uso de ferramentas de construção compatíveis com o modelo JavaBeans, os componentes podem ser combinados em *applets*, aplicações ou outros componentes [SUN2000e] e reutilizados conforme a necessidade. Os componentes JavaBeans são conhecidos como Beans e definidos da seguinte maneira:

"Um JavaBean é um componente de *software* reutilizável que pode ser manipulado visualmente numa ferramenta de programação" [SUN2000a].

Além de implementar os conceitos tradicionais de componentes de *software*, a arquitetura JavaBeans foi pensada com o objetivo de ser integrada às ferramentas de programação visual. Assim, para que sejam manipulados visualmente, os componentes expõem suas funcionalidades às ferramentas de construção visual através de *design patterns*. Os componentes são então construídos segundo determinadas regras entendidas pelas ferramentas que os utilizam, permitindo que as características dos

componentes sejam extraídas por elas através de um processo chamado introspecção. Quando os componentes seguem à risca as regras de padronização, a introspecção analisa o código objeto do componente e passa para a ferramenta todos os seus atributos. Quando isso não acontece, os atributos do componente devem ser definidos explicitamente.

As principais informações extraídas de um componente são as propriedades e os eventos que ele gera. As propriedades representam o estado e a apresentação de um componente que podem ser alterados tanto durante a fase de projeto como a de execução da aplicação. Os eventos que o componente tem são apresentados ao usuário para que ele defina as ações que devam ser executadas em função da sua ocorrência. A edição dos valores das propriedades e das ações dos eventos deve ser suportada pelas ferramentas que apresentam ao usuário editores próprios para cada caso.

Além das propriedades e eventos, os componentes JavaBeans possuem seus métodos como qualquer outra classe Java. JavaBeans não são diferentes em nenhum ponto de outras classes Java e por isso podem ser tratados como tal. Esses componentes devem apenas seguir algumas regras de padronização para que sejam entendidos pelas ferramentas de programação visual. Ainda assim, isso não é totalmente necessário, desde que as informações necessárias às ferramentas sejam especificadas explicitamente pelo programador. Isso significa que qualquer classe Java pode ser considerada um componente JavaBean.

Um exemplo irá facilitar a compreensão da estrutura de um componente JavaBean. A FIGURA 2.9 mostra a manipulação de um componente na ferramenta de programação visual JBuilder.

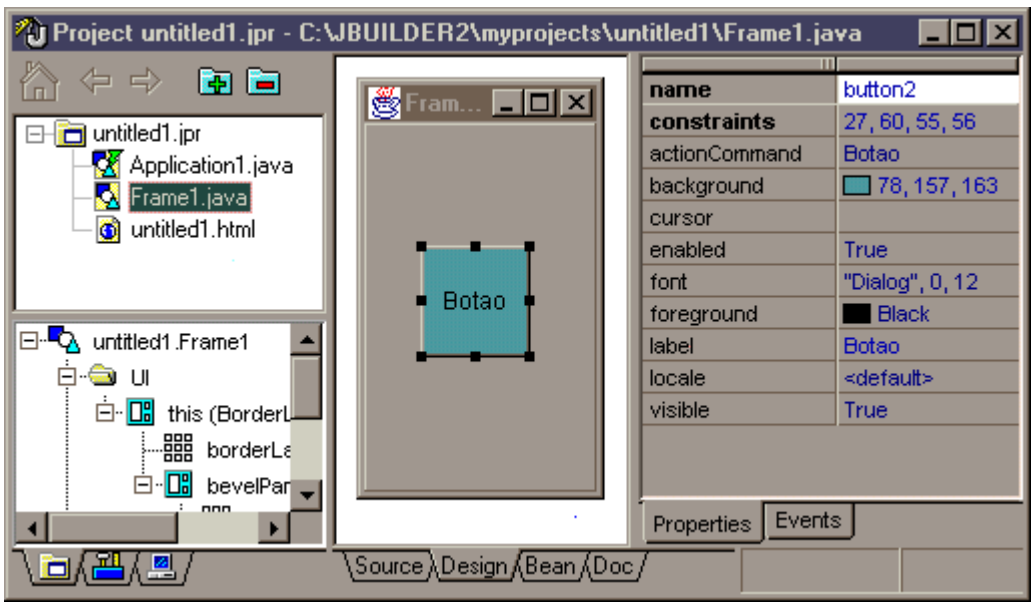


FIGURA 2.9 - Inserção de um componente botão numa aplicação.

À direita, a figura mostra a janela de edição de propriedades. Ao se selecionar o objeto, esta janela é atualizada com as informações pertinentes ao objeto em questão. As propriedades podem ser alteradas em tempo de projeto através do editor de propriedades e em tempo de execução através de chamadas de métodos no código fonte. A mesma janela possui a guia de eventos, que quando selecionada mostra os eventos que são gerados pelos objetos da classe selecionada. Para cada evento pode então ser atribuído o identificador de um método que é invocado quando o evento ocorrer.

2.5 Objetos distribuídos

Os últimos anos da computação têm sido marcados por duas tecnologias básicas que se encaixam em quase todo projeto desenvolvido nesta ciência: orientação a objeto e sistemas distribuídos. A primeira está relacionada às técnicas de projeto e desenvolvimento de *software* e a segunda segue lado a lado com as redes de computadores. Com o crescimento dessas duas tecnologias, percebeu-se que o paradigma de objetos oferece bons fundamentos para os novos desafios da computação distribuída. As noções de objeto, enraizadas no princípio de abstração de dados e na metáfora da passagem de mensagens são suficientemente fortes para estruturar e encapsular módulos de computação e também suficientemente flexíveis para se adaptarem a diversas granulosidades de arquiteturas de *hardware* e de *software* [BRI98]. Baseando-se nisso, produziu-se o que hoje se chama de "objeto distribuído", isto é, um objeto de um sistema orientado ou apenas baseado em objetos que pode estar localizado em qualquer ponto da rede e que, apesar da sua localização incerta, pode se comunicar com outros objetos através da invocação remota de método de forma muito semelhante às invocações locais.

Chegando-se a um consenso sobre os benefícios da integração entre o modelo de objetos e o de sistemas distribuídos, a questão principal que surge na implementação é como tornar acessíveis os métodos de um objeto que está localizado remotamente. Na maioria das vezes isto é resolvido por meio de um *middleware* que torne as invocações remotas quase transparentes e muito parecidas com as invocações locais. Esse agente gerencia a passagem de parâmetros, valores de retorno e erros que podem ocorrer durante a execução das operações remotas. Nesse ponto surgem diferentes arquiteturas de agentes de invocações (ou ORBs - *Object Request Broker*). Atualmente existem vários ORBs disponíveis, sendo os mais conhecidos e usados o CORBA do OMG [OMG2000], DCOM da Microsoft [MIC2000], Java RMI da SunSoft [SUN2000b] e Voyager da ObjectSpace [GLA99].

A facilidade em relação às ferramentas mais antigas de programação distribuída é o principal ponto a favor dos objetos distribuídos. A padronização de interfaces facilita o uso e adição de serviços, o que fica mais difícil em ambientes de programação de mais baixo nível, como por exemplo *sockets*. Existe ainda o RPC [SIN97] que é um avanço em relação aos *sockets* e se assemelha aos objetos distribuídos, mas que por não ser orientado a objeto peca pela falta de polimorfismo, abstração e herança de interfaces. Naturalmente todas essas facilidades têm o seu preço e em geral a eficiência da comunicação é menor quando novas abstrações são impostas aos programas (uma comparação entre o desempenho de diferentes tecnologias de objetos distribuídos e *sockets* pode ser encontrada em [HIR9?]). Entretanto, em aplicações onde a busca de desempenho não seja prioridade, os objetos distribuídos podem ser utilizados com grandes vantagens.

Quanto à modelagem de ambientes com objetos distribuídos, há controvérsias. As metodologias oriundas da engenharia de *software* costumam não fazer nenhuma distinção entre a programação centralizada com objetos e a programação com objetos distribuídos. Apenas é assumido que as invocações remotas são iguais às locais e que isso é feito de maneira transparente por algum agente de *software*. Isto parece ser razoável, mas há motivos para não se confiar totalmente nessa teoria. Guerraoui [GUE99] e Waldo [WAL94] sustentam que não é possível se alcançar a transparência total de invocação, pois sempre devem ser tratadas as exceções de falhas de comunicação ou de objetos, latência de acesso e controle de concorrência, problemas não tratados nos modelos tradicionais de orientação a objeto. A conseqüência disso é

que os modelos de programação orientados a objeto, utilizados na programação centralizada, não podem ser passados sem alterações para um ambiente distribuído. Extensões devem ser feitas para que se crie um modelo completo da realidade e assim se garanta uma implementação correta.

Assim como a orientação a objeto tradicional foi e ainda é de importância fundamental para os ambientes centralizados, os objetos distribuídos estão se estabelecendo como uma das tecnologias mais importantes para o desenvolvimento de aplicações em rede. O número cada vez mais crescente de aplicações, publicações e projetos nesta área atesta essa tendência e é mais um incentivo a que se continue nesse caminho. Na medida em que o tamanho e as necessidades de reutilização e encapsulamento das aplicações crescem, o desenvolvimento com objetos distribuídos se torna ainda mais útil e necessário.

A fim de facilitar o entendimento do funcionamento de um ambiente de objetos distribuídos, serão apresentadas algumas das principais arquiteturas existentes: CORBA, Java RMI e Voyager. Entretanto, antes de apresentá-las, vale a pena estudar os casos em que a comunicação por mensagens explícitas ainda é o meio mais adequado de interação entre duas entidades remotas, já que este tipo de comunicação também faz parte da ferramenta desenvolvida neste trabalho.

2.5.1 Comunicação por mensagens explícitas

Comunicação por mensagens explícitas é aquela onde se deve especificar claramente o conteúdo e o destino da mensagem, o que exige obviamente o conhecimento desses parâmetros. Seria de se perguntar por que utilizar trocas explícitas de mensagens para comunicação entre objetos quando há outras maneiras mais robustas e sofisticadas que realizam esta tarefa, como Voyager e RMI. Embora a troca explícita de mensagens pareça estar ultrapassada, há ainda vários motivos para se fazer uso desse meio de comunicação [FAR98]:

- as necessidades de comunicação são relativamente simples por natureza;
- o ***throughput*** de transações é crítico, demandando portanto a maior eficiência possível na comunicação;
- o escopo do sistema é limitado, de maneira que uma rápida implementação tem prioridade sobre a sofisticação e flexibilidade de projeto;
- protocolos especiais de rede precisam ser evitados (por exemplo, partes do sistema necessitam operar sob a supervisão de um ***firewall***);
- protocolos de objetos remotos simplesmente não estão disponíveis (por exemplo, uma ***applet*** num ***browser*** que não suporta RMI ou CORBA).

Algoritmos de programação já existentes que utilizam passagem de mensagens representam outra razão que sugere o uso desse tipo de paradigma. Por esses motivos, embora outros métodos ofereçam mais facilidades ao programador, a troca explícita de mensagens costuma ser ainda suportada por muitos ambientes de programação, como o HetNOS [BAR94], PVM e MPI.

2.5.2 Programação de objetos distribuídos em Java

Java é uma tecnologia nova e por isso já foi concebida se tendo em mente o seu uso em sistemas distribuídos. Várias características básicas de sua arquitetura são de

importância fundamental para o seu uso em rede: independência de plataforma, orientação a objeto, concorrência, sincronização, possibilidade de serialização de objetos e controle de segurança. A biblioteca de classes também oferece muitos recursos e abstrações para programação em rede, com classes de gerenciamento de comunicação de baixo nível (*sockets*, por exemplo) e de alto nível (URL e HTTP, por exemplo).

Além dessas características básicas, à medida que a linguagem e tudo o que está relacionado a ela evolui, novas funcionalidades para o seu emprego em rede são adicionadas. Uma das principais é a especificação RMI, onde surge o conceito de objeto distribuído propriamente dito. Com RMI é possível se ter referências locais a objetos remotos e executar chamadas a métodos desses objetos independentemente de sua localização. A estrutura de RMI trata da localização de objetos, comunicação e carga do código dos objetos através da rede de forma quase transparente para o programador.

Para construir aplicações com RMI, na implementação o programador deve inicialmente definir as interfaces para os objetos remotos, implementá-los e compilá-los como uma aplicação comum. Depois disso, deve executar um compilador especial (rmic) que gera código *stub* que funciona como um *proxy* para acesso dos clientes aos objetos remotos. Finalmente, para que a aplicação possa executar, o código dos objetos remotos deve estar acessível na rede, o que pode ser feito através de um servidor WWW.

Na execução, o objeto servidor deve antes de mais nada registrar-se no servidor de nomes RMI Registry para que os clientes possam obter as referências remotas. Tanto o registro como a localização do objeto são feitos através de código na própria implementação do cliente e do servidor. Assim que o cliente possuir uma referência remota do servidor, ele pode usá-la livremente como se fosse uma referência local (exceto pelas exceções remotas que podem ser sinalizadas).

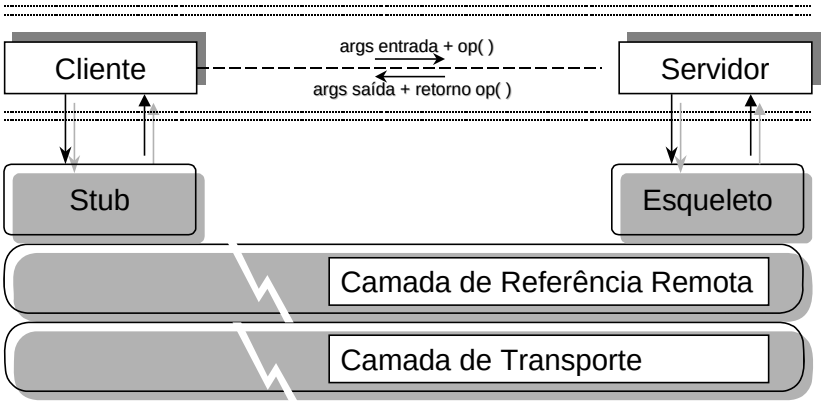


FIGURA 2.10 - Arquitetura Java RMI [HER98].

Como se vê na FIGURA 2.10, um programa desenvolvido em Java RMI apresenta na base de sua arquitetura a camada de transporte responsável pela transferência de dados de baixo nível. Acima dela, outra camada trata do mapeamento das referências locais a objetos remotos utilizadas nos clientes. A parte *stub* permite o acesso do cliente às referências remotas e o esqueleto disponibiliza a interface do objeto servidor. Para o usuário, praticamente tudo isso é transparente.

Java RMI oferece apenas uma estrutura básica de objetos distribuídos em Java. Não satisfeitos com este conjunto de funcionalidades, empresas privadas e pesquisadores acadêmicos desenvolvem camadas mais abstratas sobre a plataforma básica de Java. Os projetos mais conhecidos são o Voyager [GLA99], Concordia

[MIT2000] e IBM Aglets [IBM2000]. Esses ambientes foram construídos principalmente para a implementação de aplicações de agentes distribuídos, procurando simplificar ao máximo o uso de objetos distribuídos além de oferecer novos serviços, como por exemplo um serviço de diretório universal.

Uma característica também muito importante para o uso de Java em sistemas de objetos distribuídos é a possibilidade de integração com outras tecnologias. Java possui todas as ferramentas necessárias para que seja integrada naturalmente com CORBA que, como será visto a seguir, é uma tecnologia padrão para objetos distribuídos.

2.5.3 CORBA

Com o crescente número de aplicações executando em rede, o desejo natural de integração dessas aplicações esbarrou na questão de interoperabilidade: como fazer interagir aplicações projetadas para e que executam em diferentes plataformas de *hardware*, sistema operacional e linguagem de programação? Pensando nisso, desde 1991 o *Object Management Group* (OMG) desenvolve o padrão *Common Object Request Broker Architecture* (CORBA) para promover a portabilidade, interoperabilidade e compatibilidade entre diferentes plataformas [OMG2000]. Criado em 1989, o OMG é um consórcio sem fins lucrativos, composto por centenas de entidades (principalmente empresas e universidades), que se dedica a promover a teoria e a prática da tecnologia de objetos para o desenvolvimento de sistemas de computação distribuída.

CORBA segue a *Object Management Architecture* (OMA). OMA é uma visão em alto nível de um ambiente distribuído completo que descreve as interações entre os componentes e as camadas CORBA. O principal componente desta arquitetura é o *Object Request Broker* (ORB), que oferece a infra-estrutura com os mecanismos básicos de comunicação para que os componentes possam interagir num ambiente heterogêneo, distribuído e independente de plataforma. Diferentes ORBs podem se comunicar através do protocolo *General Inter-ORB Protocol* (GIOP). O GIOP possui uma adaptação para TCP/IP (protocolo *Internet Inter Orb Protocol* - IIOP), possibilitando que diferentes ORBs se comuniquem através da Internet.

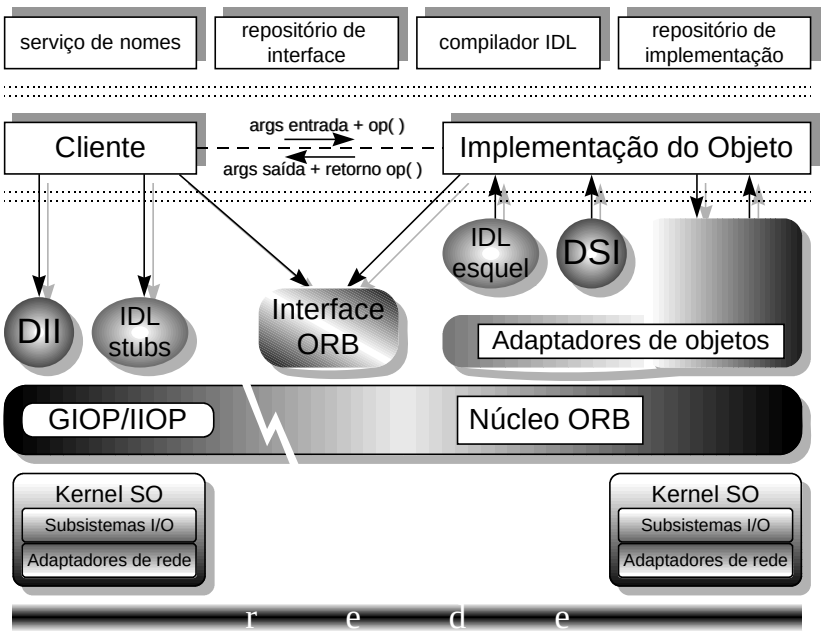


FIGURA 2.11 - Arquitetura CORBA.

O ORB se responsabiliza por todo o mecanismo necessário para encontrar a implementação do objeto, prepará-la para uma requisição e enviar os dados correspondentes. O ORB chama automaticamente no cliente e no objeto servidor serviços de segurança segundo a CORBA *Security Architecture*. Caso a aplicação deseje por si só realizar esse serviço, ela deve fazer chamadas diretas a serviços de segurança.

Em CORBA, um cliente pode requisitar um serviço de um objeto distribuído sem se preocupar com a sua localização, em que sistema operacional ele está executando ou ainda em que linguagem de programação ele foi codificado. Para isso, antes de se disponibilizar um serviço, as interfaces dos objetos precisam ser definidas através da *Interface Definition Language* (IDL), linguagem de definição orientada a objeto que suporta múltipla herança de interfaces. As interfaces e métodos IDL são mapeados nas construções da linguagem em que será programado o serviço, como por exemplo C++ ou Java. As interfaces podem ser acrescentadas ao serviço *Interface Repository*, que representa e organiza as informações de uma interface, permitindo acesso em tempo de execução a essas informações.

Antes de fazer uma requisição, o cliente lê a definição IDL da interface do servidor e escolhe entre utilizar a IDL *stub* (*Static Invocation Interface* - SII - dependente da interface do objeto alvo) ou a *Dynamic Invocation Interface* (DII - independente do objeto alvo) para efetuar sua requisição. As interfaces estática e dinâmica satisfazem à mesma semântica, impedindo que o receptor da mensagem saiba de que maneira a requisição foi feita.

Os serviços devem ser registrados com o ORB e podem ser instalados em qualquer máquina. Os clientes se conectam a um serviço usando um agente de localização ou explicitamente informando o nome do servidor, que pode ser obtido através de um servidor de nomes. Além de objetos desenvolvidos pelo usuário, uma série de serviços é disponibilizada por CORBA, como transações, concorrência e eventos [OMG98].

Há várias implementações de CORBA disponíveis para as mais diversas plataformas. Não existe ainda uma implementação padrão, mas em geral as existentes atualmente conseguem interoperar sem maiores problemas utilizando os protocolos que o próprio OMG define.

2.5.4 Voyager

Voyager [GLA99] é o nome dado a uma linha de produtos destinados à computação distribuída da qual fazem parte os produtos *Voyager ORB*, *Voyager ORB Professional*, *Voyager Management Console*, *Voyager Security*, *Voyager Transactions* e *Voyager Application Server*. Desenvolvida totalmente em Java pela empresa norte-americana ObjectSpace, Voyager tem sido usada como ferramenta de base na computação com objetos distribuídos pela facilidade e recursos que oferece para transportar para um sistema distribuído uma aplicação tradicional orientada a objeto.

O principal produto da linha é o Voyager ORB. O Voyager ORB é um agente de invocações de objetos que suporta simultaneamente CORBA e RMI e possui um serviço de nomes e diretório universais que permite a interoperabilidade entre aplicações desenvolvidas com diferentes tecnologias de objetos distribuídos. Além disso, suporta agentes móveis, comunicação em grupo e diversos tipos de mensagens (síncronas, assíncronas e futuras). Um ponto forte de Voyager é a possibilidade de adequar a um ambiente distribuído classes Java já existentes sem a necessidade de que sejam modificadas, liberando assim o programador da exigência da posse do código fonte.

A arquitetura universal de Voyager (FIGURA 2.12) permite que diferentes tecnologias de objetos sejam utilizadas simultaneamente. Ela é dividida em diversas camadas que podem ser substituídas conforme as necessidades de cada usuário. Por exemplo, na camada que representa o modelo de objetos, pode ser utilizado tanto CORBA, como RMI, DCOM e o próprio Voyager (VNM).

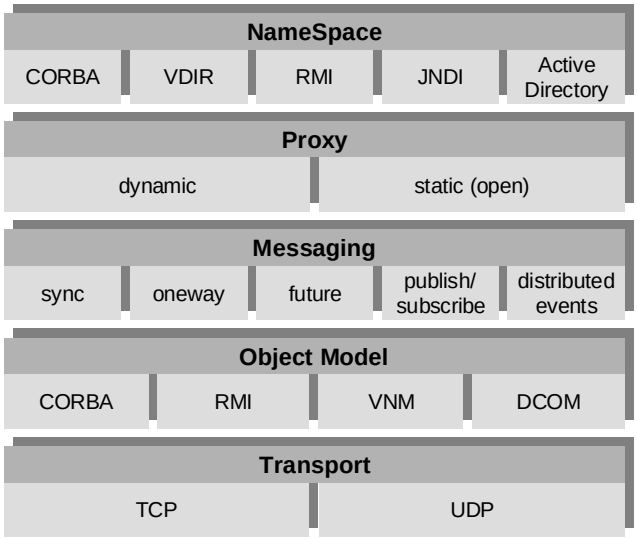


FIGURA 2.12 - Arquitetura Voyager ORB [GLA99].

Além da invocação remota de métodos e da arquitetura de comunicação universal, outras características importantes de Voyager são a construção remota de objetos, carga dinâmica de classes e tratamento de exceções remotas. Realiza também a coleta de lixo distribuída e suporta persistência de objetos.

Todas as características apresentadas são importantes, mas o ponto forte de Voyager é a simplicidade que há em se construir um programa distribuído, como mostrado no exemplo que segue [GLA99a]. A fim de criar um sistema cliente-servidor em Voyager, o programador deve executar os seguintes passos:

1. Criar a interface Java.

```
public interface IAdder extends com.objectspace.voyager.IRemote {
  int add(int x, int y);
}
```

FIGURA 2.13 - Interface Java do servidor.

2. Criar a implementação da interface.

```
public class Adder implements IAdder {
  public int add(int x, int y) {
    System.out.println("add " + x + " and " + y);
    return x + y;
  }
}
```

FIGURA 2.14 - Implementação da interface do serviço.

3. Criar o servidor

```
import com.objectspace.voyager.*;

public class VoyagerServer {
  public static void main(String[] args) {
    try {
       Voyager.startup("8000"); // start up on port 8000
```

```
        IAdder adder = new Adder();
        Namespace.bind("VoyagerAdder", adder); // bind to local NS
        System.out.println("server ready");
     } catch (Exception exception) {
        exception.printStackTrace();
     }
  }
}
```

FIGURA 2.15 - Implementação do servidor.

4. Criar o cliente

```
import com.objectspace.voyager.*;

public class VoyagerClient {
  public static void main( String[] args ) {
    try {
      Voyager.startup(); // start up on random port
      IAdder adder = (IAdder)
                Namespace.lookup ("//localhost:8000/VoyagerAdder");
      System.out.println( "3 + 4 = " + adder.add( 3, 4 ) );
    } catch (Exception exception) {
      exception.printStackTrace();
    }
    Voyager.shutdown();
  }
}
```

FIGURA 2.16 - Implementação do cliente.

5. Executar o servidor e o cliente

```
> java VoyagerServer
server ready
add 3 and 4
> java VoyagerClient
3 + 4 = 7
```

FIGURA 2.17 - Execução do servidor e do cliente.

O exemplo mostrado na seqüência da FIGURA 2.13 até a FIGURA 2.17 demonstra a facilidade oferecida por Voyager para a disponibilização de objetos num sistema distribuído. As alterações que devem ser feitas no código são mínimas e outras operações como recompilação também não são necessárias na maioria dos casos. Além dessas facilidades, a quantidade e a qualidade dos serviços oferecidos pela ferramenta e as opções de interoperabilidade são outras razões pelas quais Voyager é utilizada atualmente por empresas do mundo inteiro como base para a computação com objetos distribuídos.

2.6 Ferramentas RAD

A última análise das mais importantes tecnologias aplicadas neste trabalho trata das ferramentas RAD. RAD (*Rapid Application Development*) é um termo dado a determinadas tecnologias cujo objetivo principal é tentar reduzir o tempo de construção do *software*. Entre essas tecnologias estão as ferramentas RAD que aplicam esse conceito nos ambientes de desenvolvimento para a geração de aplicações num curto espaço de tempo. Essas ferramentas são utilizadas predominantemente em mercados onde a competição é acirrada e o sucesso de um produto depende muito da freqüência de novos lançamentos. A preocupação com a implementação das ferramentas RAD

começou a partir do fim da década de 90 e, assim como a busca da produtividade e da qualidade do *software*, tornou-se uma das "modas" da indústria de *software* [CAR95].

Para chegarem ao seu objetivo e desenvolverem aplicações rapidamente, em geral os ambientes RAD reduzem o tempo pela eliminação ou automatização de determinadas atividades como codificação e documentação [CAR95]. A execução concorrente de tarefas e a reutilização do *software* são outros fatores importantes para a economia no tempo de desenvolvimento. Em termos práticos, muitas ferramentas RAD existentes atualmente obtêm esse título pela facilidade com que são usadas e com que permitem a rápida disponibilização de protótipos. Para isso, possibilitam a construção rápida de interfaces gráficas através de programação e manipulação visual de componentes, combinadas com geração automática de código.

O motivo de se fazer uma breve introdução às ferramentas RAD nesse capítulo vem do fato de que a ferramenta desenvolvida neste trabalho possui muitas funcionalidades baseadas nesses ambientes RAD. A programação visual com componentes, reutilização e geração automática parcial de código são as mais importantes. Embora o objetivo principal aqui não seja o de desenvolver aplicações em tempo mínimo, as facilidades encontradas nas ferramentas de desenvolvimento rápido podem ser usadas com grande utilidade na construção de aplicações distribuídas.

Apesar de atualmente existirem muitas ferramentas RAD e elas estarem sendo utilizadas com sucesso, elas podem nem sempre dar o mesmo resultado a todas as áreas de mercado [CAR95]. Por suas características, as ferramentas RAD são mais importantes aos mercados de competição sensível à velocidade com que novos produtos são lançados, o que deixa determinados casos mais dependentes a outros fatores que não o tempo de desenvolvimento.

2.6.1 Ferramentas de programação visual em Java

Em vista do sucesso das ferramentas de programação visual para o ambiente Windows como Delphi e Visual Basic, esse tipo de ferramenta começou a ser disponibilizado também para Java pouco tempo depois do aparecimento das primeiras versões do JDK da Sun. Assim como acontece com as outras ferramentas RAD, essas ferramentas representam os principais ambientes de desenvolvimento para a linguagem em questão para aplicações de pequeno e médio porte.

Existem muitas ferramentas RAD de programação visual em Java. A arquitetura padrão própria de componentes JavaBeans é um incentivo ao aparecimento de novas ferramentas desse tipo. As principais disponíveis comercialmente são Borland JBuilder, Microsoft Visual J++, IBM VisualAge e Symantec Visual Café. A estrutura geral de todas essas ferramentas segue mais ou menos o mesmo padrão, sendo constituída da seguinte maneira (como demonstra a FIGURA 2.18):

- barra de ferramentas com botões para as operações mais freqüentemente executadas;
- paleta de componentes;
- editor de propriedades e eventos: quando um objeto é selecionado, as suas propriedades são exibidas e podem ser editadas, assim como a correspondência entre métodos e eventos;
- editor visual da interface gráfica e editor do código textual.

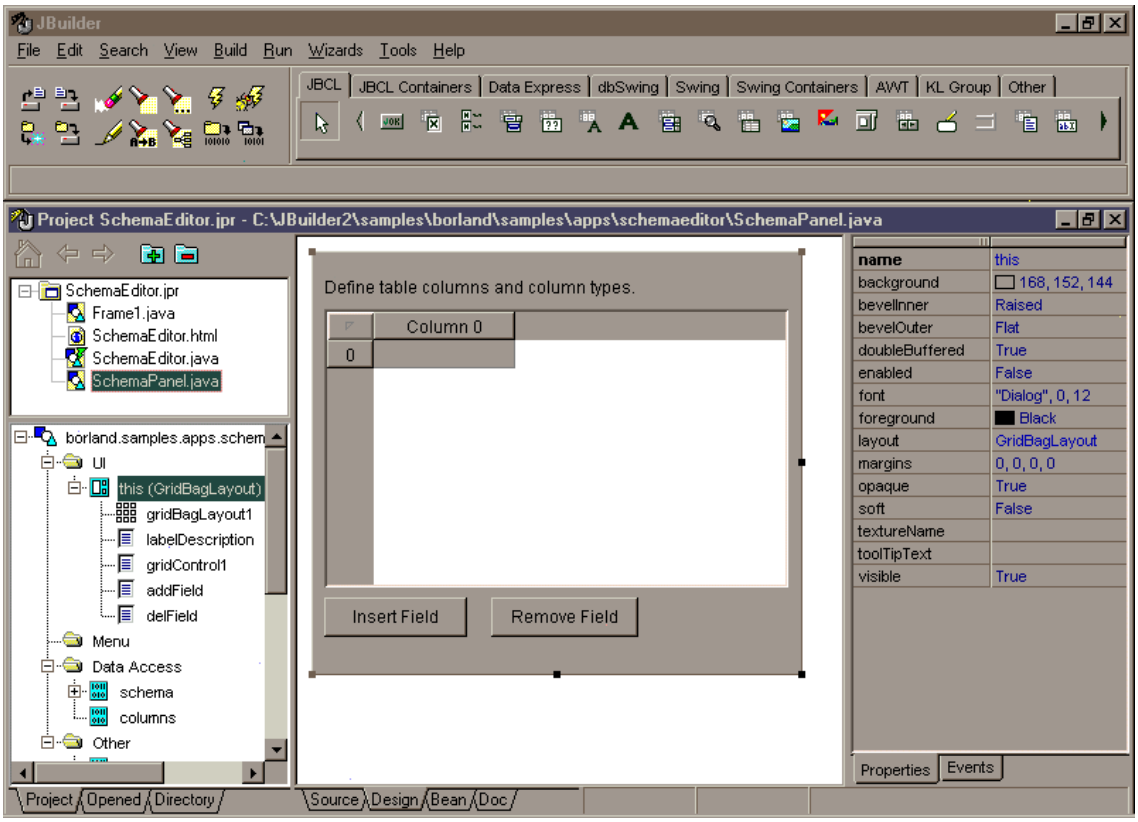


FIGURA 2.18 - Interface de programação da ferramenta JBuilder.

Nessas ferramentas, a edição gráfica é integrada com a edição textual dos programas. Portanto, quando alguma alteração é feita na parte gráfica, as mudanças são refletidas também no código textual. Esta é uma das principais características que dá o título de RAD a essas ferramentas, pois, para construir a interface do programa, o usuário lida apenas com objetos gráficos e a ferramenta se encarrega de gerar o código da implementação.

Outra característica importante dessas ferramentas é a possibilidade delas incorporarem ao seu ambiente de desenvolvimento novos componentes. Componentes JavaBeans desenvolvidos por terceiros podem ser integrados à paleta de componentes e utilizados da mesma forma que os componentes disponibilizados com a ferramenta.

Essas ferramentas fazem atualmente grande sucesso entre os desenvolvedores de *software*, o que prova o poder e a simplicidade da sua metodologia de trabalho. Entretanto, seu foco de ação ainda são as aplicações isoladas. Embora seja possível por meio delas criar aplicações cliente-servidor, o cliente e o servidor devem ser construídos um de cada vez. Elas ainda não são capazes de representar uma aplicação paralela ou distribuída segundo uma visão geral onde todos os elementos concorrentes possam ser identificados clara e imediatamente. É justamente esse problema que o ambiente de programação proposto neste trabalho procura resolver. Seu objetivo é oferecer uma maneira mais dirigida e adequada à implementação de aplicações concorrentes, onde o programador tenha uma visão global da sua aplicação e ao mesmo tempo aproveite todas as vantagens e facilidades que uma boa ferramenta de desenvolvimento possa oferecer, como programação visual, reutilização de componentes e geração automática de código.

2.7 Resumo do capítulo

Este capítulo tratou das principais tecnologias utilizadas na construção desta ferramenta. A programação visual é importante porque os gráficos possuem melhor capacidade de representar entidades bidimensionais como os programas concorrentes do que os programas textuais. Dentro dela, o tipo de representação mais comum em programas concorrentes é o modelo de grafos, utilizado na grande maioria das ferramentas de programação visual paralela e distribuída.

Quanto ao usuário, ele estará diretamente envolvido com a programação orientada a eventos e a estruturação das aplicações em termos de componentes. Essas duas tecnologias lhe permitirão desenvolver objetos com grande independência entre si, facilitando a reutilização de código e a extensão das aplicações.

A comunicação entre os objetos localizados em diferentes máquinas pode ser feita tanto por trocas de mensagens explícitas como por invocação remota de métodos. Neste último caso, têm-se os objetos distribuídos propriamente ditos, que se comunicam através da rede de forma quase transparente para o programador.

Ultimamente, as ferramentas RAD têm feito muito sucesso entre os programadores, principalmente para o desenvolvimento de aplicações centralizadas. Muitas de suas características são voltadas para a engenharia de *software* e por isso são capazes de prover auxílio na construção de *software* com produtividade e qualidade.

3 Descrição da Ferramenta

Neste capítulo será apresentada a ferramenta de programação visual para aplicações com objetos distribuídos em Java proposta neste trabalho e suas características fundamentais. Inicialmente serão estudadas as metas que se pretendem atingir com a realização deste trabalho e em seguida serão descritas as principais funcionalidades da ferramenta. O modelo de programação visual, uma das partes mais importantes deste ambiente, também será explicado neste capítulo.

3.1 Objetivos e requisitos da ferramenta

Os objetivos e requisitos de uma ferramenta de programação para ambientes paralelos e distribuídos devem obviamente ser buscados nas necessidades de seus programadores. Muitas delas são necessidades comuns aos desenvolvedores de aplicações centralizadas e são resultado dos princípios básicos de engenharia de *software*. Dentre os principais objetivos e requisitos, destacam-se:

- **programação orientada a objeto**: a possibilidade de empregar os conceitos de encapsulamento, modularização, herança e polimorfismo é importante no desenvolvimento de qualquer tipo de aplicação, seja ela concorrente ou não;
- **programação visual**: como foi afirmado na seção 2.1.2, a visualização é muito importante em programação concorrente. Além disso, é difícil de se construir e de se manter um programa orientado a objeto sem construções visuais. Os vários métodos de análise e projeto orientado a objeto utilizam alguma espécie de diagrama [GRU95];
- **facilidade de uso**: a programação concorrente é por si só mais complexa do que a programação centralizada, o que torna a facilidade de uso um aspecto ainda mais crítico;
- **modelo de programação abstrato**: responsabilizando-se pela parte de programação de mais baixo nível e oferecendo ao usuário um modelo mais abstrato, a ferramenta livra o usuário de tarefas cuja execução é fonte provável de erros (embora a maioria dos erros ainda resida na fase de projeto do *software* [FIS90]). A geração automática de código é uma das formas de abstrair os níveis mais baixos de implementação;
- **flexibilidade**: apesar de utilizar uma linguagem visual para a programação (o que costuma reduzir a flexibilidade), a geração de código fonte textual e o controle do usuário sobre ele mantêm a flexibilidade da programação textual tradicional;
- **rapidez de desenvolvimento**: facilitar o uso da ferramenta é uma maneira de aumentar a velocidade de desenvolvimento e por conseguinte a produtividade;
- **portabilidade**: em sistemas distribuídos, a execução de programas em ambientes heterogêneos é uma necessidade, o que exige o suporte a diferentes plataformas de *hardware* e de *software*;
- **opções comuns**: um programa pode ter muitos parâmetros cujos valores podem ser assumidos previamente sem que isso o afete de maneira decisiva. Sendo assim, a ferramenta pode iniciar muitas variáveis com o valor mais provável ou

outro qualquer que leve a um estado consistente, evitando que o programador tenha que defini-los manualmente um a um. Essa possibilidade diminui bastante o tempo de manipulação da linguagem visual, principalmente no desenvolvimento de protótipos;

- **reutilização de código**: com a utilização de um padrão de componentes de *software*, a ferramenta pode tanto utilizar componentes desenvolvidos em outras ferramentas como aproveitar posteriormente em outras aplicações parte do trabalho feito nela própria;
- **extensão do sistema**: novos componentes podem ser adicionados à ferramenta como uma alternativa para a solução de problemas por ela ainda não atacados.

Esses objetivos foram definidos com base na experiência e análise tanto das qualidades como das deficiências das ferramentas usadas atualmente na programação paralela e distribuída. Além disso, alguns objetivos, como orientação a objeto e reutilização de código, são herança dos ambientes de programação para ambientes centralizados, onde estes conceitos se encontram muito mais desenvolvidos. Nestas ferramentas, como as ferramentas RAD, tais conceitos são aplicados exaustivamente, oferecendo um melhor retorno ao programador em termos de produtividade e qualidade de *software*.

De uma certa forma relacionados, os principais objetivos desta ferramenta são a facilidade de uso e a programação visual. Como já foi falado anteriormente, a programação visual é muito útil na especificação e visualização de programas concorrentes. A capacidade que os gráficos têm de representar estruturas bidimensionais é o que a torna tão útil neste contexto. Portanto, é interessante que a ferramenta permita a descrição visual da concorrência e da comunicação, oferecendo uma linguagem suficientemente poderosa para expressar um bom número de aplicações e que ao mesmo tempo seja simples, evitando que descrições complexas demais prejudiquem a compreensão do programa.

Assim, a programação visual é utilizada para expressar, em nível mais abstrato, a concorrência e a comunicação. Para que seja possível especificar também os detalhes dos métodos da lógica da aplicação, a ferramenta deve combinar representações visuais com textuais [GRU95]. Algumas ferramentas de programação visual utilizam uma sintaxe adicional de anotação dos objetos com o fim de especificar os detalhes que não podem ser expressos visualmente. Essa abordagem, todavia, não garante a flexibilidade de que muitos usuários podem necessitar, pois a sintaxe utilizada para especificar a anotação é comumente apenas uma forma de estruturar, em mais alto nível, a aplicação concorrente e seus elementos segundo um modelo de programação que em geral é incompleto. Com elas se pode definir a estrutura das mensagens e o comportamento dos processos que depois serão passados a um gerador de código. Este, por sua vez, traduzirá esses dados para uma forma de código textual, escrito em uma linguagem de programação concorrente ou uma linguagem de propósito geral (C, C++, Java) que implementa as funções de paralelismo e distribuição através de bibliotecas. O usuário perde a flexibilidade quando o gerador de código esconde dele a tradução da fase de alto nível para a de baixo nível e não permite que ele trabalhe sobre esse código fonte gerado.

A FIGURA 3.1 apresenta um exemplo contendo a descrição de um nodo no grafo de uma aplicação desenvolvida com a ferramenta VisualProg [SCH96]. Nesse modelo, o usuário deve programar através da declaração de regras de ativação (STARTING_RULES), que controlam as atividades a serem desempenhadas em função

da chegada de mensagens, e regras de propagação de dados (ROUTING_RULES), que definem os dados a serem enviados conforme o estado dos dados locais.

```
DECLARATIONS
      int ok;
END

STARTING_RULES
      init init2 ;
      on receive
            B : { v_print("recebi B\n"); };
            tamanho: { v_print("Recebi tamanho = %i\n", tamanho); };
      end;
      B, tamanho: multiplica;
END

PROC init2
      ok = 0;
END

PROC multiplica
      int         i;

      for (i = 0; i < tamanho; i++)
            D.B[i] = B[i] * v_get_id();
      D.instancia = v_get_id();
      ok = 1;
END

ROUTING_RULES
(ok):{
            v_print("Instancia %i mandou D\n", v_get_id());
      } : D;
END
```

FIGURA 3.1 - Anotação dos nodos na ferramenta VisualProg.

Neste modelo, o usuário tem pouca liberdade para controlar os aspectos de concorrência e comunicação, pois está restrito ao modelo apresentado pela ferramenta. A vantagem obtida é que essa visão abstrata facilita a compreensão do funcionamento do programa, pois ele já está estruturado segundo um modelo de programação concorrente, o que não acontece numa linguagem de programação geral como C ou C++.

Além da conseqüente perda de flexibilidade, outro problema enfrentado pelo programador na utilização de uma sintaxe adicional de anotação é a necessidade de seu estudo e conhecimento. Como não existe um padrão nessa área, cada ferramenta utiliza sua própria linguagem de anotação, o que acaba dificultando o uso entre ferramentas.

Outro requisito importante para a ferramenta é a portabilidade das aplicações. A geração de programas para ambientes distribuídos termina por criar essa necessidade. Tal problema é resolvido neste trabalho através do ambiente Java, que oferece portabilidade entre as principais plataformas de *hardware* e de *software* de forma natural. Além disso, a possibilidade de comunicação através de *sockets* permite que uma aplicação se comunique com praticamente todos os sistemas existentes atualmente. Por causa da Internet, os protocolos TCP e UDP são um padrão de fato, sendo largamente utilizados.

A utilização do modelo de componentes de *software* serve a dois propósitos principais: reutilização de código e extensibilidade. Como apresentado na seção 2.4, as vantagens da programação com componentes são muitas e por isso são buscadas na grande maioria dos ambientes de programação mais recentes.

A eficiência das aplicações geradas, embora sempre importante, não é um dos principais objetivos desta ferramenta. Na programação paralela a eficiência é um ponto crítico do desenvolvimento, o que leva os programadores a realizar esforços maiores em programação de baixo nível com o objetivo de otimizar o código e aumentar a velocidade final de execução. Por outro lado, na programação distribuída, a questão da eficiência é muitas vezes deixada de lado em favor de uma maior transparência e facilidade de implementação.

3.1.1 A que ambiente se destina essa ferramenta?

Antes de se criar uma ferramenta, deve-se ter uma definição clara de onde ela será usada. Do seu contexto fazem parte tanto os tipos de aplicações como os tipos de usuários, o que forma o cenário que servirá de base para que o projetista da ferramenta analise as necessidades existentes e crie meios para satisfazê-las.

3.1.1.1 Aplicações

Como vem sendo mencionado ao longo de todo este texto, o alvo desta ferramenta são as aplicações paralelas e distribuídas, sendo dada uma ênfase especial a estas últimas. A este conjunto pertencem programas que podem executar em qualquer tipo de rede, seja ela local ou global, bastando para isso que o usuário possua os direitos de acesso às máquinas que deseja utilizar.

A especificação dos programas segue o paradigma da programação concorrente explícita, onde é responsabilidade do programador fazer a decomposição paralela, sincronização e comunicação. Essa escolha representa uma das razões pelas quais se está utilizando a programação visual neste ambiente de programação, pois ela é ainda mais útil quando o paralelismo e a distribuição devem ser definidos explicitamente.

Ao contrário das ferramentas especializadas em programação paralela onde se desenvolvem predominantemente aplicações mestre-escravo, as aplicações mais comuns a serem desenvolvidas com esta ferramenta são as aplicações cliente-servidor. Elas podem ser implementadas tanto com objetos distribuídos utilizando invocação remota de métodos e criação remota de objetos, ou apenas com objetos localizados em diferentes endereços interagindo através de sistemas de comunicação de mais baixo nível, como *sockets*. As necessidades e restrições das aplicações é que irão determinar a sua forma de implementação.

Esta ferramenta também pode ser utilizada para o desenvolvimento de componentes de comunicação para clientes que estejam sendo construídos em outras ferramentas. Por exemplo, pode-se supor que exista um servidor de transações de banco de dados cujo acesso seja feito somente através de invocação remota de método suportada por Voyager ou RMI. Naturalmente, dispondo-se das bibliotecas requeridas, é possível se implementar essa aplicação em qualquer ferramenta de programação Java. Entretanto, a parte da comunicação será muito mais facilmente implementada na ferramenta proposta neste trabalho, pois todos os componentes de acesso a esses sistemas já estão prontos. O programador utiliza então esta ferramenta para desenvolver o componente de comunicação e o inclui em outra ferramenta que possui mais recursos

de desenvolvimento centralizado, como assistentes para geração de interface gráfica e depuração de código. Assim como podem ser implementados os componentes de comunicação, qualquer tipo de programa cliente, onde o servidor disponibiliza a sua interface através de um dos meios de comunicação suportados pela implementação da ferramenta, também pode ser implementado.

Uma característica importante relacionada aos tipos de aplicações é o seu modelo de eventos. O modelo de eventos é bastante apropriado a redes globais, onde o tempo de resposta é indefinido (podendo ser bastante grande). Com o modelo de eventos, pode-se evitar que um programa fique bloqueado esperando pela resposta a uma mensagem, permitindo que ele execute outras tarefas enquanto espera a notificação de chegada. Quando a notificação acontecer, ele estará pronto para ler a mensagem. O mesmo acontece com o envio das mensagens. Num sistema normal com transmissão confiável (com confirmação), o programa fica bloqueado enquanto envia a mensagem, pois ele tem que passar pelas fases de conexão, transferência de dados e desconexão. No modelo de eventos, o programa executa a chamada *send*, que na verdade apenas coloca a mensagem numa fila para envio posterior. As mensagens dessa fila são processadas e enviadas numa *thread* separada, o que evita o bloqueio da *thread* principal. Quando a mensagem foi enviada com sucesso, a *thread* que executou o *send* é notificada do evento.

Outras atividades também são mais simples de serem programadas através de eventos. A detecção de erros de comunicação é uma tarefa complicada na programação tradicional. Na maioria dos casos, um erro é percebido somente quando a tarefa não foi concluída com sucesso. Com eventos, a implementação pode gerar um evento e notificar a aplicação no momento em que o erro acontece. O código que descobre a ocorrência dos erros executa de forma transparente para o usuário.

Para ampliar a abrangência de tipos de aplicações, uma das principais características e objetivos dessa ferramenta é a flexibilidade. Dessa forma, muitas classes de aplicações podem ser desenvolvidas. Caso a ferramenta não disponibilize na sua implementação inicial todas as características necessárias, em muitos casos a criação de novos componentes pode ser uma solução para o problema. Por isso, programas que necessitem de recursos especiais como dados globais e *locks* globais também podem se beneficiar desta ferramenta, já que estes componentes foram a ela adicionados.

Resumindo, esta ferramenta pode ser utilizada no desenvolvimento tanto de aplicações cliente-servidor completas como apenas de componentes de comunicação. Nesses dois casos, a ferramenta oferece grande flexibilidade, permitindo que o programador tenha controle fino sobre os seus programas. Além disso, através dos componentes de *software*, o desenvolvedor obtém facilidades a respeito da reutilização do seu código e do de outros programas. Como exemplos de aplicações que podem ser construídas com esta ferramenta, pode-se citar qualquer tipo de aplicação cliente-servidor, como acesso a banco de dados, transações de comércio eletrônico, bate-papo e controle de estoque distribuído, entre outras.

3.1.1.2 Usuários

Com relação aos usuários, esta ferramenta não se destina àqueles que sejam totalmente principiantes. Não é seu objetivo ser utilizada por pessoas que desejem programar aplicações distribuídas como se o estivessem fazendo com aplicações centralizadas. Isso é buscado em algumas ferramentas de programação visual para

ambientes paralelos, que apresentam ao usuário fortes abstrações que tentam aproximar a programação concorrente da programação seqüencial. Entretanto, na maioria das vezes, pelo menos parte da especificação da decomposição paralela necessita ser feita explicitamente, o que exige o conhecimento deste assunto por parte do usuário. Esta tarefa está longe de ser algo simples [PUR98], fazendo com que programadores restritos a ambientes centralizados, sem conhecimento nenhum de programação concorrente, não estejam capacitados para esta tarefa. A conseqüência disso é que somente programadores de ambientes distribuídos que já possuam uma certa experiência se aventuram a utilizar essas ferramentas de maneira realmente efetiva.

Esse objetivo de tentar aproximar ao máximo a programação concorrente da programação seqüencial não existe neste trabalho porque a meta principal aqui é o desenvolvimento de aplicações distribuídas. Uma aplicação paralela é semelhante a uma aplicação centralizada em mais aspectos do que uma aplicação distribuída. Embora um programa paralelo seja formado por vários elementos de execução, ele se inicia num determinado instante e após um período limitado de tempo é encerrado. Nesse meio tempo o programador não precisa ter o conhecimento de que houve divisão de tarefas ou não. Por isso, as ferramentas podem fazer a decomposição dos programas de forma transparente. Em aplicações distribuídas, entretanto, o mesmo não pode ser feito sempre. Em um ambiente cliente-servidor, é comum que o usuário seja obrigado a localizar e informar explicitamente o servidor, o que impede que a distribuição seja totalmente transparente.

O tratamento diferente que é dado às aplicações paralelas pode também ser constatado intuitivamente pela existência de compiladores paralelizadores, mas inexistência de compiladores "distribuidores". Isto ocorre porque o conhecimento do servidor é algo que deve ser buscado fora do programa. Na programação paralela, todos os elementos estão dentro do mesmo programa, isto é, tanto as partes que realizam o trabalho como as partes que o requisitam são de conhecimento de todos. Numa aplicação distribuída, o servidor deve ser informado explicitamente e dificilmente pode ser deduzido. Além disso, como já foi dito, antes do início de uma aplicação paralela e após o final dela não há partes executando. Isto parece óbvio, mas numa aplicação distribuída isso acontece somente com os clientes. O servidor, de acordo com a sua própria definição, pode ficar executando indefinidamente, de tal maneira que o início e o fim da aplicação como um todo não possam ser delimitados.

Outra questão que exige programadores experientes em sistemas distribuídos na programação é o fato de que as aplicações distribuídas são em geral desenvolvidas separadamente. Em programação paralela, o mais comum é que a mesma pessoa ou grupo seja o responsável pela programação tanto da parte mestre como da parte escrava. Portanto, se uma ferramenta de programação como um compilador paralelizador tiver à sua disposição o cenário completo, ela terá maior liberdade para controlar as interações entre os diferentes objetos do sistema. Em aplicações distribuídas, o mais comum é que a programação do cliente e do servidor sejam feitas por entidades sem vínculo entre si. Isso exige que se definam interfaces de acesso bem claras, o que não pode ser feito sem uma boa experiência prévia em sistemas distribuídos.

Além de tudo isso, após um estudo das ferramentas de programação visual existentes, constatou-se que muito poucas são realmente utilizadas na prática. Entre outros motivos, concluiu-se que o principal deles é a falta de flexibilidade. Como os programadores de aplicações concorrentes são em geral experientes em linguagens textuais, eles estão acostumados com a flexibilidade que elas oferecem e sentem muita falta dela na sua ausência. Eles reconhecem os benefícios da programação visual, mas

preferem continuar se valendo de uma ferramenta que vem lhe dando resultados satisfatórios há algum tempo (embora esses resultados pudessem ser melhores). A impossibilidade de aproveitar o código já desenvolvido em outras ferramentas também é um empecilho ao uso dessas novas ferramentas de programação visual.

Portanto, decidiu-se projetar a ferramenta de forma que ela se voltasse aos programadores mais experientes, oferecendo-lhes principalmente a flexibilidade que eles apreciam. Acredita-se que por este motivo as ferramentas do tipo de Delphi e JBuilder fazem tanto sucesso, já que embora se possa programar visualmente a interface gráfica nessas ferramentas, a possibilidade de continuar programando diretamente no código textual garante a flexibilidade procurada por muitos usuários.

3.2 Estrutura geral da ferramenta

A estrutura geral da ferramenta é muito semelhante à estrutura das ferramentas RAD brevemente apresentadas no capítulo anterior. Ela é praticamente a mesma porque todas essas ferramentas lidam com o desenvolvimento através de componentes, havendo para isso tanto um editor visual como um editor textual utilizados simultaneamente na programação das aplicações. Devido à larga aceitação daquelas ferramentas, concluiu-se que uma estrutura semelhante, como a apresentada na FIGURA 3.2, seria agradável aos usuários não somente pela funcionalidade mas também pela experiência que eles porventura possam ter tido com o uso daquelas outras ferramentas.

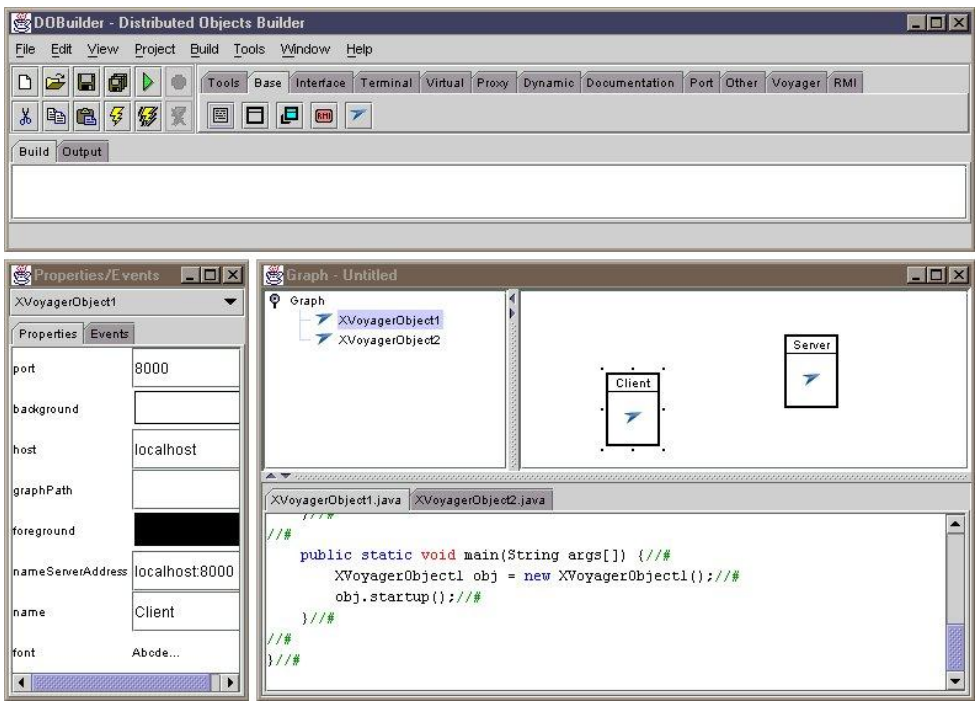


FIGURA 3.2 - Interface gráfica da ferramenta.

Sendo assim, as principais partes da ferramenta são:

- **janela principal**: contém a barra de menus e de ferramentas, a paleta de componentes e uma área de texto contendo as mensagens da ferramenta e a saída das aplicações;
- **janela de edição do grafo**: formada pelo editor de grafos, pela visualização da estrutura do grafo e pelos editores de texto onde o código Java dos nodos é editado;

- **janela de propriedades e eventos**: apresenta editores para as propriedades dos objetos e possibilita a definição dos métodos tratadores dos eventos gerados pelos objetos.

Estas são as partes principais da ferramenta. Existem, entretanto, outros módulos que também possuem a sua importância, mas que não serão apresentados aqui por uma questão de espaço. A partir de agora seguem os detalhes dos módulos citados acima.

3.2.1 Janela principal

Esta é a interface gráfica apresentada inicialmente ao usuário. Os comandos do usuário são feitos a partir da sua barra de menus e de ferramentas. A saída dos comandos de compilação e de execução é capturada e apresentada na parte inferior da janela.

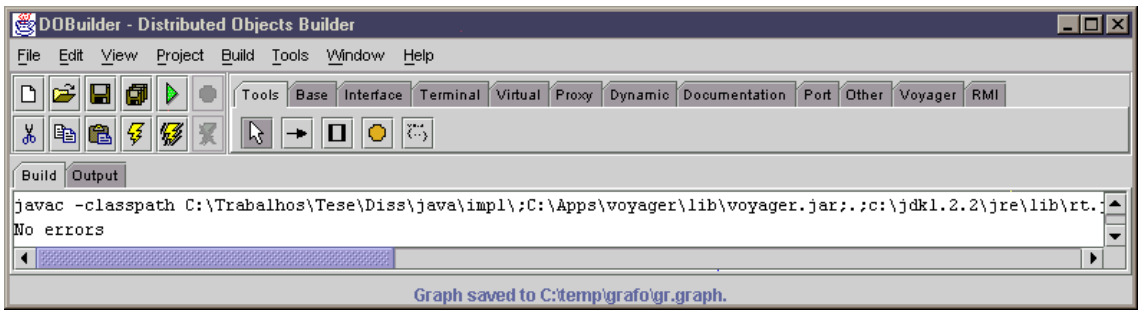


FIGURA 3.3 - Janela principal.

A parte mais importante da janela principal é a paleta de componentes, que contém os tipos de objetos que podem ser usados numa aplicação. Como mostra a FIGURA 3.3, ela é dividida em várias guias contendo cada uma os ícones correspondentes aos componentes, podendo inclusive ser configurada para que sejam adicionadas novas guias e novos componentes.

3.2.2 Janela de edição do grafo

Nesta janela é editada de forma gráfica e interativa a estrutura de um grafo. É composta por três partes: o editor gráfico, a árvore da estrutura do grafo e o editor textual do código fonte em Java dos nodos.

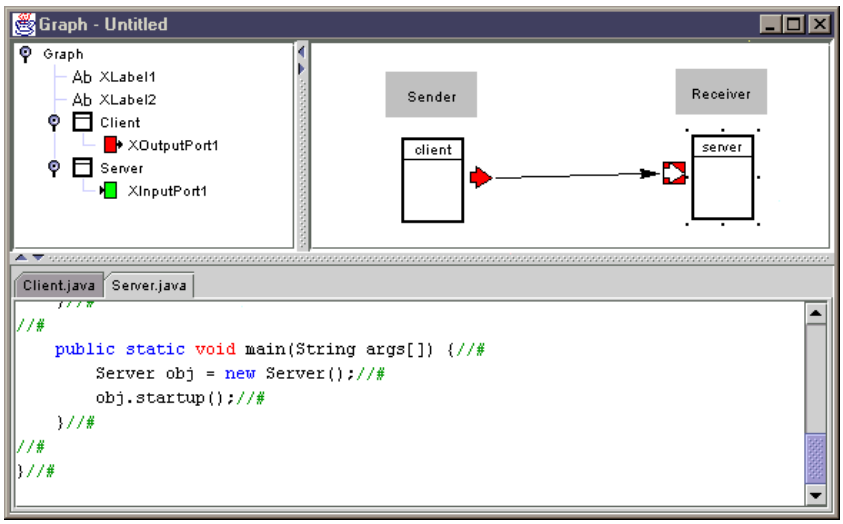


FIGURA 3.4 - Janela de edição do grafo.

Cada janela dessas contém tudo o que é necessário para a edição de um grafo, o que permite que várias janelas sejam abertas e assim mais de um grafo possa ser editado ao mesmo tempo (útil principalmente quando uma aplicação é composta por vários grafos). A FIGURA 3.4 mostra um grafo na janela de edição de grafos. O editor gráfico fica à direita e acima, a árvore de estrutura do grafo à sua esquerda e o editor textual, abaixo.

O editor de grafos é utilizado para construir o grafo contendo os objetos e os relacionamentos (FIGURA 3.5). Os objetos a serem inseridos são buscados na paleta de componentes da janela principal.

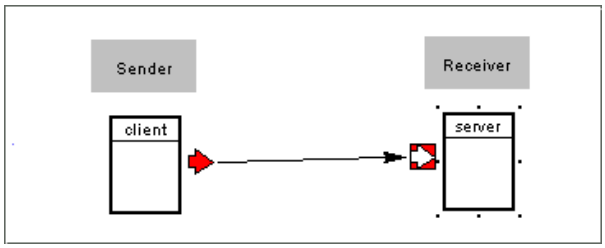


FIGURA 3.5 - Editor de grafos.

À medida que o grafo vai sendo construído, a sua estrutura vai sendo atualizada na árvore da estrutura do grafo (FIGURA 3.6), que contém os objetos do grafo organizados de forma hierárquica. Dessa maneira é mais fácil perceber quais objetos fazem parte da aplicação e quais os relacionamentos de composição que há entre eles.

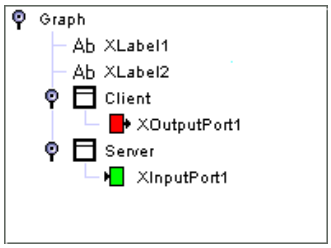


FIGURA 3.6 - Árvore da estrutura do grafo.

Como também pode ser visto na FIGURA 3.6, o objeto possui um ícone ao lado do seu identificador, indicando a que classe ele pertence. Os nomes dos objetos apresentados nesta estrutura hierárquica são utilizados na geração de código e podem ser alterados conforme o desejo do usuário. São iniciados com o nome da classe à qual o objeto pertence, seguido por um número de seqüência e devem ser únicos dentro do projeto.

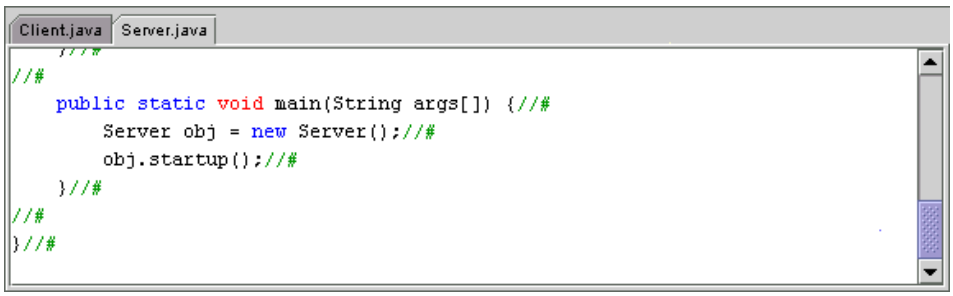


FIGURA 3.7 - Janela de edição do código textual.

Aqueles objetos cujo código pode ser editado recebem uma janela de edição na parte inferior. Esta seção exibe o código fonte em Java gerado pela ferramenta para um determinado nodo e oferece a possibilidade de que o usuário faça as suas próprias alterações, como mostra a FIGURA 3.7.

Cuidados especiais devem ser tomados pelo usuário para que ele não interfira no código gerado pela ferramenta. O usuário pode inserir linhas de código entre as linhas geradas, mas não lhe é permitido alterar nem remover uma linha de programa que tenha sido gerada pela ferramenta. Maiores detalhes sobre a geração de código são expostos nas seções 3.3.4 e 4.9.

3.2.3 Janela de propriedades e eventos

Cada componente utilizado nesta ferramenta possui propriedades e métodos tratadores de eventos que podem ser editados em tempo de projeto. Esta janela oferece os editores para as propriedades e para os identificadores dos métodos correspondentes a cada evento (FIGURA 3.8).

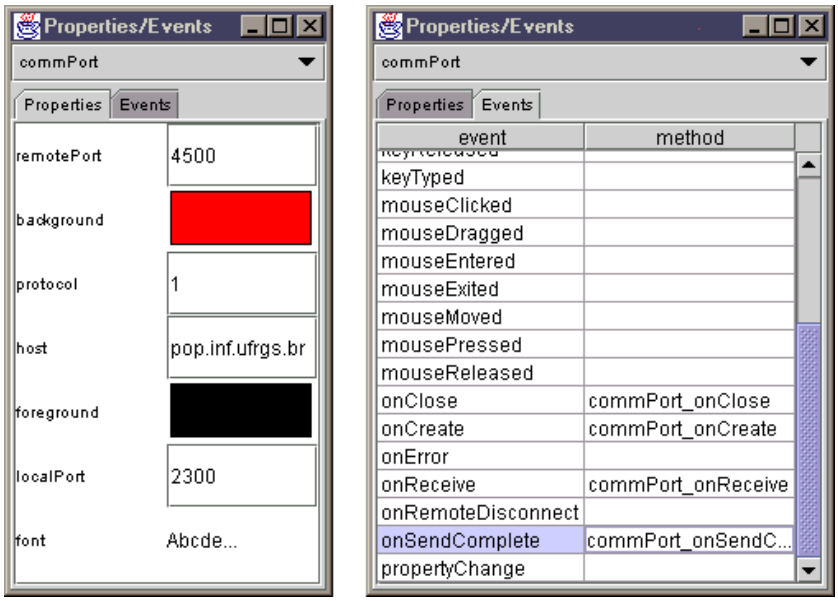


FIGURA 3.8 - Janela de propriedades e eventos.

Quando um objeto é selecionado no editor gráfico ou na janela da hierarquia do projeto, a janela de propriedades e eventos é atualizada com os atributos correspondentes a tal objeto. Dependendo do componente, as alterações nas propriedades e nos identificadores dos eventos são refletidas imediatamente no código fonte gerado.

3.3 Programação de aplicações

Esta seção tratará basicamente do ciclo de desenvolvimento de aplicações nessa ferramenta. Serão descritos os principais passos e a seqüência necessários para a criação de uma aplicação qualquer, procurando-se dar uma idéia mínima do uso desta ferramenta.

3.3.1 Processo de desenvolvimento de aplicações

Qualquer aplicação editada nesta ferramenta é composta por pelo menos um grafo. Em linhas gerais, o desenvolvimento apresenta a seguinte seqüência, que também pode ser vista de forma esquemática na FIGURA 3.9.

- criação de um projeto, associando-se um grafo a este projeto;
- criação de um ou mais grafos da aplicação;

- geração de código para os grafos e o projeto;
- compilação do código gerado com um compilador java (javac, por exemplo);
- execução da aplicação com um interpretador java (java) em cima de um ambiente distribuído (Voyager ou RMI).

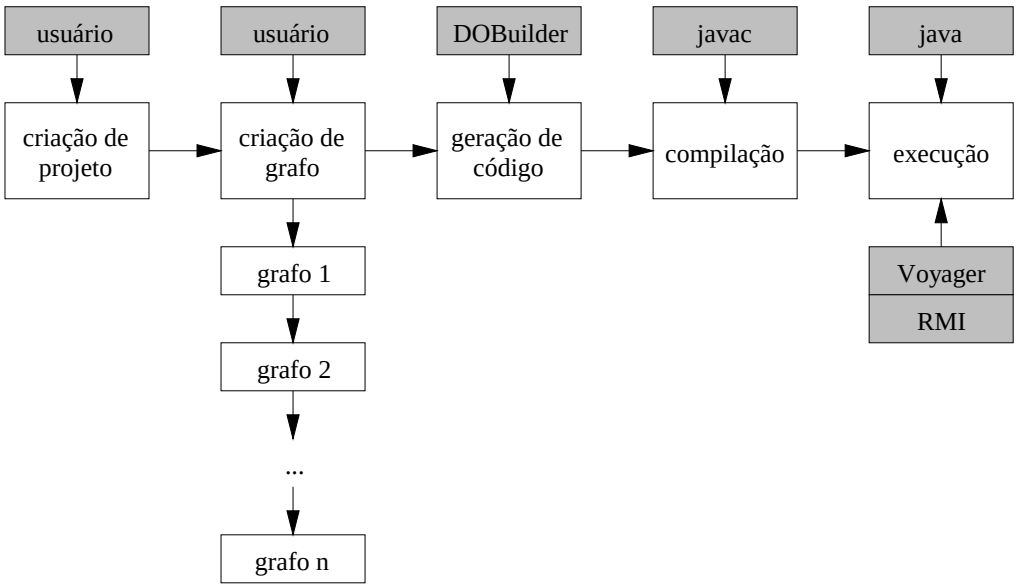


FIGURA 3.9 - Processo de desenvolvimento de aplicações.

Para cada aplicação deve ser criado um projeto contendo o seu grafo principal, que será o ponto de entrada para a execução da aplicação. Os grafos são independentes de projeto e por isso podem ser utilizados em diferentes projetos.

A criação dos grafos da aplicação é feita interativa e graficamente através do editor de grafos. O programador escolhe os componentes que irá utilizar na paleta de componentes e os manipula visualmente no editor de grafos. À medida que a estrutura do grafo é definida, ela é visualizada em forma de árvore na janela do projeto. Há também a definição das propriedades e dos métodos de tratamento dos eventos através da janela de propriedades e eventos. Ao mesmo tempo é exibido o código dos nodos básicos e dos nodos de interface, de maneira que ele esteja diretamente acessível ao programador e assim possa ser complementado com o código da aplicação.

A geração e compilação dos objetos dos grafos podem ser feitas diretamente através da ferramenta, mas a execução depende do tipo de aplicação que foi gerada. Conforme o ambiente de execução escolhido, a execução deve ser feita manualmente, porque ela às vezes envolve o lançamento de programas em diferentes máquinas, o que nem sempre é possível de se fazer através da ferramenta. Nesses casos, há a necessidade de execução de comandos remotos, que podem não estar disponíveis ou liberados nas redes de computadores utilizadas.

3.3.2 Modelo de programação orientado a eventos

A programação orientada a eventos é um dos pontos fundamentais desta ferramenta. O modelo de programação proposto apresenta um fluxo de execução onde o andamento das atividades da aplicação é controlado por meio de eventos gerados externamente aos objetos. É um tipo de programação diferente do método tradicional seqüencial com chamadas de métodos e por isso exige um certo tempo de adaptação.

Apesar disso, já é bastante conhecido e utilizado em muitos ambientes de programação (principalmente na programação de interfaces gráficas).

A programação visual se combina muito bem com o paradigma de programação orientada a eventos, o que produz muitas ferramentas de programação visual que definem os seus objetos com base nos eventos que eles geram e recebem. Nesse paradigma, a comunicação entre dois objetos se dá apenas através de eventos. Assim, os objetos são definidos de forma independente entre si, facilitando a sua representação visual, já que a visualização é mais fácil de ser feita quando os objetos possuem um acoplamento fraco entre si. Objetos com muitos relacionamentos entre si são difíceis de se representar visualmente.

Nesta ferramenta, os eventos são o meio de comunicação utilizado entre os componentes que se relacionam localmente (não confundir com a comunicação entre os objetos distribuídos, que se dá por meio de chamadas remotas e trocas de mensagens). Foi escolhida a arquitetura de componentes JavaBeans, onde os componentes são modelados segundo propriedades, métodos e eventos. Um objeto que agrega um componente se registra como ouvinte dos eventos que este componente dispara. Os eventos que o objeto receber vão dirigir as suas ações, que por sua vez podem gerar outros eventos e assim desencadear ações em outros objetos ou até mesmo nele próprio.

Como está colocado na seção 3.1.1.1, o paradigma de eventos facilita a programação distribuída, principalmente onde não há sincronismo e o tempo de resposta das operações é indefinido. Além de facilitar a programação, essa estratégia também aumenta a concorrência, pois evita esperas e bloqueios desnecessários.

3.3.3 Componentes de *software*

Outra característica importante desta ferramenta é o desenvolvimento das aplicações em termos de componentes, permitindo que os objetos sejam reutilizados e também que novos componentes sejam adicionados às aplicações de forma natural. Como os componentes são modelados segundo um padrão já estabelecido e definido (JavaBeans), componentes que tenham sido desenvolvidos em outras ferramentas podem ser utilizados aqui e vice-versa.

Na implementação da ferramenta, os componentes devem estar disponíveis para que os usuários possam incluí-los nas aplicações. Isso é feito através de uma paleta de componentes, onde eles estão divididos por função. É possível também incorporar novos objetos a essa paleta, o que estende a funcionalidade do ambiente de programação.

3.3.4 Geração de código

A linguagem visual é independente da geração de código. Um mesmo tipo de representação visual de um programa pode ser usado para a geração de código que execute em diferentes ambientes de execução. A semântica do programa visual continua a mesma, mas a sua implementação pode ser bastante variável. Por esse motivo, é interessante que a geração de código não seja uma ação imutável e controlada totalmente pelo criador da ferramenta, mas sim personalizada pelo seu usuário. A ferramenta pode oferecer geradores de código padronizados que se apliquem a uma boa parte dos programas que sejam criados, mas não pode se restringir a somente esses tipos de programas.

Em vista da necessidade de personalização da geração de código, o gerador de código desta ferramenta não está implementado na ferramenta, mas sim nos objetos do usuário. Essa independência entre a geração de código e a ferramenta é importante quando a geração de código necessite ser personalizada pelo usuário.

Embora possa num primeiro momento parecer uma carga adicional ao usuário, que seria o responsável pela geração de código, não é isso o que acontece. A ferramenta oferece os geradores de código padrão que são usados quando o usuário quiser gerar os tipos de aplicações já suportados pela ferramenta. Quando ele desejar criar outros tipos de aplicações, basta que ele desenvolva novos geradores de código ou apenas faça modificações nos já existentes. Esta possibilidade, entretanto, não impede que o desenvolvedor da ferramenta continue provendo novos tipos de objetos, o que facilita também a atualização no lado do usuário, pois a ferramenta não precisa ser substituída por uma nova versão caso apenas um novo ambiente de execução venha a ser suportado.

A independência entre o editor visual e o gerador de código torna a ferramenta muito mais modular. Entre essas duas entidades há apenas uma interface que indicará as ações que deverão ser tomadas pelo gerador de código quando o usuário executar os seus movimentos. Todas as ações que venham a alterar o código da aplicação deverão ser notificadas ao gerador de código para que ele execute as medidas correspondentes. As ações que devem ser implementadas pelos geradores de código são as seguintes:

- inserção, remoção e alteração do nome de qualquer objeto no editor de grafos;
- mudança no valor das propriedades e nos identificadores dos métodos tratadores de eventos;
- estabelecimento de relacionamento entre os objetos;
- geração de código para criação dos objetos.

Os detalhes dessas ações são mostrados na seção 4.9.1, que trata da implementação do gerador de código. Essas ações são desencadeadas pelo usuário durante a edição do grafo, momento em que o editor chama os métodos correspondentes no gerador de código, tornando consistentes as representações textuais e visuais.

A localização da implementação da geração de código no objeto do usuário não permite somente que vários ambientes de execução sejam suportados, mas também que uma mesma aplicação seja composta por objetos que façam parte de diferentes ambientes de execução. A comunicação através de portas de comunicação TCP/IP possibilita que um objeto executando num ambiente distribuído Java puro possa se comunicar com outro objeto executando num ambiente distribuído Voyager. Para que isto aconteça, basta que cada objeto seja programado de forma a adequá-lo ao ambiente de execução alvo.

3.3.5 Integração com outras ferramentas

Uma das características mais desejáveis de qualquer *software* é a possibilidade de que ele se integre da maneira mais fácil possível com soluções de outros fabricantes. Essa possibilidade é mais um motivo para que ela seja empregada por um número ainda maior de usuários, pois estes terão a certeza de que os seus investimentos já feitos em outras plataformas serão preservados, pelo menos em parte. Integrando-se as ferramentas se pode aproveitar os benefícios de cada uma, superando as deficiências específicas e chegando a uma ferramenta geral mais poderosa.

A principal característica que permite a integração com outras ferramentas é a utilização do padrão de componentes JavaBeans. Isto permite que todos os ambientes de programação que suportem JavaBeans possam interagir através da troca de componentes. Como esses componentes são padronizados, um componente que foi desenvolvido para e por uma ferramenta pode ser integrado normalmente a outra.

No caso desta ferramenta, os componentes podem ser usados de duas maneiras distintas para realizar a interface com outros ambientes de programação. No primeiro caso, sabe-se que esta ferramenta não foi criada com o objetivo de gerar interfaces gráficas. Por isso, para integrá-las às aplicações geradas aqui, componentes de interfaces podem ser desenvolvidos nas ferramentas visuais para programação em Java que possuem essa capacidade. No outro caso, o aproveitamento de componentes ocorre no sentido inverso. Como o ponto forte desta ferramenta é a comunicação em ambientes distribuídos, componentes de comunicação podem ser desenvolvidos aqui e integrados às ferramentas de programação visual em Java.

A integração entre as ferramentas é a forma de se criar um ambiente completo de programação. Construir um sistema de bate-papo, por exemplo, utilizando somente uma das ferramentas é uma tarefa complicada. Se for usada a ferramenta JBuilder, será fácil construir a interface gráfica, mas a comunicação entre os objetos será mais difícil de ser implementada. Se for utilizada a ferramenta proposta aqui, a parte de comunicação será programada facilmente, enquanto a interface gráfica deve ser feita codificando-se diretamente. A integração das duas através de componentes JavaBeans permite que os recursos de cada uma sejam aproveitados para produzir uma aplicação que preencha a todos os requisitos. Na prática, o programador desenvolve a interface gráfica como um componente JavaBean no JBuilder e a incorpora a um nodo básico no grafo da aplicação distribuída. Outra maneira é desenvolver o componente de comunicação nesta ferramenta e depois integrá-lo à ferramenta de programação visual em Java. As duas maneiras são possíveis e representam a forma implícita de se integrar essas ferramentas.

3.4 Modelo de programação visual

A linguagem visual é a base da programação visual. Com o objetivo de ser flexível, a linguagem deve oferecer um determinado conjunto de elementos visuais que sejam capazes de representar os comportamentos mais diversos de um programa. Tal objetivo é atingido através da especificação de diferentes elementos visuais e de suas relações, o que constitui o projeto da linguagem visual.

Esta seção apresentará a estrutura da linguagem visual utilizada na programação com esta ferramenta. Inicialmente, será dada uma visão geral sobre todos os seus objetos e, em seguida, uma descrição mais minuciosa dos mesmos.

3.4.1 Linguagem visual

A linguagem visual é um dos aspectos mais importantes do modelo de programação desta ferramenta. Todo o cuidado deve ser tomado a fim de que os vários requisitos de uma linguagem desse tipo sejam satisfeitos e assim um programa possa ser especificado de maneira clara e correta através dela. Os vários requisitos já foram expostos anteriormente e mostram a dificuldade existente na criação de uma linguagem visual. As decisões muitas vezes subjetivas que devem ser tomadas no momento do projeto da linguagem podem acabar se traduzindo em empecilhos ao usuário. Por isso, é difícil se afirmar com antecedência se uma determinada linguagem visual será amplamente usada e realmente útil. Na maioria das vezes, chega-se a essa conclusão

somente quando muitos testes com várias classes de usuários tiverem sido feitos. Da mesma forma, não são raras as vezes em que os resultados obtidos não são os esperados.

A linguagem visual aqui apresentada representa uma aplicação distribuída como um grafo dirigido. Este tipo de representação é utilizado em muitas ferramentas de programação visual paralela e distribuída - HeNCE [BEG94], CODE 2 [NEW92][NEW93], VPE [NEW95], VisualProg [SCH96], GRADE [KAC97a], Meander [WIR94], P-RIO [LOQ98], Enterprise [CHA91][WIL93], Paralex [DAV96], Tracs [BAR95], PVMBuilder [PED99] - e é considerada a forma mais adequada de se representar uma estrutura concorrente [BRO94]. Neste esquema, a aplicação é um conjunto de nodos e arcos, onde os nodos indicam os elementos distribuídos e os arcos dão informações sobre o relacionamento entre tais elementos. A semântica referente aos arcos não é rígida e depende tanto do modelo geral de programação escolhido como dos tipos de objetos envolvidos no relacionamento. Há linguagens visuais de programação paralela onde o arco indica fluxo de dados (CODE 2, VPE, VisualProg, GRADE, P-RIO, Tracs) e outras onde a mesma representação se traduz por fluxo de controle (HeNCE, Meander, Paralex, Enterprise). Neste trabalho, o significado do relacionamento dependerá dos tipos de objetos envolvidos. Por exemplo, numa relação entre uma porta de comunicação de entrada e uma de saída, o arco indica comunicação por troca explícita de mensagem. Em outro caso, quando o arco conecta um objeto cliente a um objeto que exporta um método remoto, a interação indicará que o objeto origem está chamando um método executado remotamente no objeto designado pelo nodo destino do arco. Para identificar com clareza o tipo de interação, a representação visual do arco também varia.

Portanto, assim como num grafo, a especificação de uma aplicação nesta ferramenta é feita por nodos e arcos. Entretanto, apenas nodos e arcos simples não são suficientes para expressar as soluções para os principais problemas que envolvem objetos distribuídos. A linguagem proposta baseou-se então no modelo de grafos e o enriqueceu com outras construções a fim de que um maior número de aspectos pudessem ser modelados.

3.4.2 Nodo

O nodo, juntamente com o arco, corresponde à parte mais importante da linguagem visual. Os tipos de nodos (ou objetos do grafo) existentes nessa linguagem visual são os seguintes (FIGURA 3.10):

- **nodo básico**: representa a unidade básica de distribuição personalizada pelo usuário. O objeto designado por este nodo terá o seu código editado pelo programador com a lógica da aplicação para tal objeto;
- **nodo de interface**: nodo que descreve a interface de acesso a um objeto distribuído;
- **nodo virtual**: representa um objeto externo ao sistema e acessado pela aplicação, como por exemplo um servidor de envio de mensagens de correio eletrônico. A ferramenta não gera código para este objeto, mas através do nodo virtual os objetos da aplicação tomam informações sobre como ter acesso ao objeto real;
- **nodo terminal**: é um nodo básico ou de interface em formato de código objeto. Não pode, portanto, ter seu código editado;

- **componente**: módulo de *software* adicionado a um nodo básico com o objetivo de executar alguma função específica. A ligação entre um componente e o nodo básico a que ele pertence é feita através do registro e notificação de eventos;
- **componente procurador**: tipo especial de componente acoplado a um nodo básico para que este possa invocar métodos em objetos remotos;
- **componente para criação dinâmica**: também é um tipo especial de componente agregado a um nodo básico e tem a finalidade de prover a esse nodo básico a capacidade de criar dinamicamente outros objetos distribuídos;
- **método**: objeto adicionado a nodos básicos e nodos de interface com o intento de representar visualmente porções do código desses objetos;
- **nodo de interface de grafo**: define um objeto do grafo que será visível fora dele;
- **chamada a grafo**: a fim de facilitar a visualização de aplicações com um grande número de objetos, um objeto de chamada a grafo encapsula uma estrutura composta por vários objetos num único objeto no programa visual.

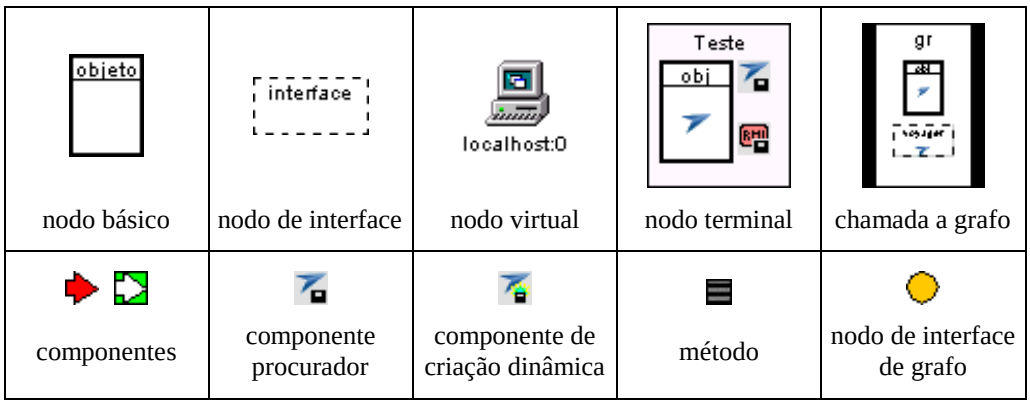


FIGURA 3.10 - Tipos de objetos da linguagem visual.

Os objetos apresentados até agora e que podem ser vistos na FIGURA 3.10 são usados diretamente na especificação do programa. Embora o objetivo da linguagem seja a programação, outros tipos de objetos podem ser adicionados com propósitos diferentes, mas que de uma certa forma estão ligados à programação. Assim, utilizados principalmente para a documentação e apresentação visual do programa, os seguintes tipos de elementos visuais foram acrescentados (FIGURA 3.11):

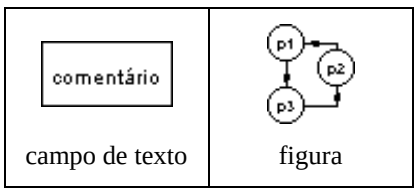


FIGURA 3.11 - Objetos para documentação e apresentação visual.

- **campo de texto**: ao lado de um nodo ou objeto qualquer, pode dar uma descrição rápida do seu comportamento ou objetivo;
- **figura**: com a mesma finalidade do campo de texto, pode também ser útil na explicação do funcionamento do programa.

Com todos esses tipos de objetos é possível construir um número razoável de classes de aplicações distribuídas. A escolha desses tipos de objetos foi baseada no estudo e análise de algumas aplicações que revelaram a sua necessidade. Não se pretende, portanto, afirmar que esta linguagem seja capaz de expressar qualquer programa distribuído. Seu objetivo é especificar as questões mais importantes nessa área, como invocação de método, criação de objetos e definição de interfaces. Além disso, uma estruturação modular da linguagem que permitisse uma implementação baseada em componentes foi outro objetivo perseguido neste projeto.

Feita esta introdução, todos os tipos de objetos da linguagem visual apresentados acima serão descritos a seguir com detalhes.

3.4.3 Nodo - Nodo básico

O nodo básico é a principal unidade de distribuição da aplicação. Sua função é a de conter o código da lógica da aplicação para um objeto distribuído. O programador pode editar o código do objeto correspondente ao nodo básico e também adicionar componentes cujos eventos serão tratados por tal objeto. A FIGURA 3.12 apresenta três nodos básicos, cada um específico para um ambiente de execução. Os objetos menores são componentes, que serão vistos na seção 3.4.7.

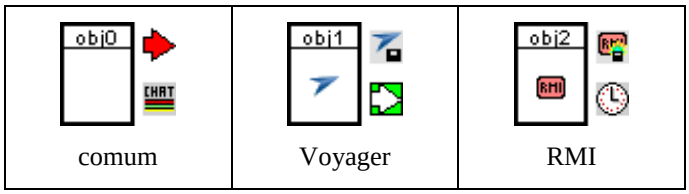


FIGURA 3.12 - Nodos básicos.

O nodo básico é um objeto autônomo e implementa essa característica através de um corpo com código que é executado logo após a sua criação. Esse conceito é semelhante ao conceito de *body*, apresentado em muitas linguagens de programação concorrente orientada a objeto que seguem o modelo de objetos ativos autônomos [BRI98]. Exemplos dessas linguagens são Eiffel// [CAR93] e POOL [AME87].

Os objetos da aplicação representados pelos nodos básicos estão sempre localizados em espaços de endereçamento distintos e devem se valer de mecanismos especiais de comunicação para a realização de uma tarefa comum (chamadas de métodos remotos ou trocas de mensagens explícitas). A única exceção é a possibilidade de compartilhamento de memória entre dois nodos básicos, o que será visto mais tarde na seção 3.4.17.

O código da aplicação contido pelo nodo básico é composto por duas fontes: pela parte inserida pelo usuário e pela parte gerada pela ferramenta. A geração de código comandada pela ferramenta acontece à medida que o usuário faz alterações no programa visual. Quando algum componente é adicionado a um nodo básico ou quando alguma propriedade é alterada, por exemplo, o seu código é modificado imediatamente. Mais informações sobre a geração de código podem ser encontradas nas seções 3.3.4 e 4.9, sendo que esta última já trata da sua implementação.

3.4.3.1 Propriedades

Assim como os outros objetos, o nodo básico é um componente que possui as suas propriedades e métodos, descritos na TABELA 3.1.

TABELA 3.1 - Propriedades de um nodo básico.

propr./método	**descrição**
máquina	máquina em que está ou será localizado
gerador de código	gerador de código chamado pela ferramenta em função das ações tomadas pelo usuário
nome	nome como é conhecido na rede (é colocado no servidor de nomes)
nodo pai	referência ao objeto que o criou
caminho do grafo	identificação do grafo ao qual pertence o nodo básico
servidor de nomes	endereço do servidor de nomes que armazena os dados dos objetos da aplicação distribuída
`startup`	método de iniciação chamado por quem cria o objeto
`shutdown`	método de finalização chamado pelo usuário quando o objeto deve ser encerrado
`run`	método chamado dentro do método `startup`, logo após a iniciação. É o ponto de entrada do código do usuário

A linguagem visual não especifica qual deve ser o comportamento do nodo básico. Ela apenas define algumas propriedades que devem ser apresentadas para que ele seja encarado dentro de um programa como um objeto desse tipo. Assim, os tipos de nodos básicos não são definidos pela ferramenta e podem ser criados livremente pelo usuário. Para que um usuário crie um nodo básico, basta que ele dê a um objeto as propriedades apresentadas acima. Ao incluir um objeto desse tipo no grafo da aplicação, a ferramenta irá detectar que ele é um nodo básico e passará a tratá-lo como tal.

A TABELA 3.1 apresenta as características que um nodo básico deve possuir. Elas são necessárias para que todas as características do modelo possam ser implementadas. No decorrer do texto, especialmente no capítulo de implementação, a função de cada uma dessas propriedades será melhor esclarecida.

A representação visual do nodo básico está na FIGURA 3.12. Já que cada objeto pode ter sua representação configurada conforme o gosto do usuário, essa representação não é imposta, mas apenas sugerida. Todos os nodos básicos implementados neste trabalho se assemelham a essa representação.

3.4.4 Nodo - Nodo de interface

Os nodos de interface representam as interfaces de acesso para invocação de métodos remotos. Um objeto que disponibiliza métodos remotos é obrigado a colocá-los numa interface para que os outros objetos distribuídos tenham conhecimento da forma de como ter acesso a eles. Esse acesso é feito posteriormente por meio de componentes procuradores (vide seção 3.4.8) ou de criação dinâmica (vide seção 3.4.9).

A FIGURA 3.13 mostra o relacionamento típico existente entre um nodo de interface, um nodo básico e um componente procurador. A representação mostra que o objeto indicado pelo nodo básico *server* implementa a interface definida no nodo de interface *voyager*. Sendo assim, todos os métodos que estão naquela interface devem ser implementados pelo objeto distribuído. A conexão entre o nodo de interface e os

componentes procuradores e de criação dinâmica indicam que eles representam tal interface.

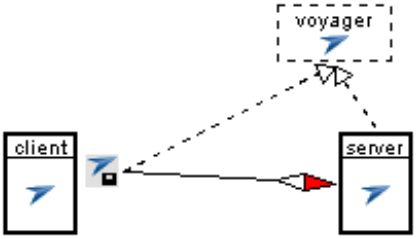


FIGURA 3.13 - Nodo de interface *voyager* conectado a um nodo básico e a um nodo procurador.

Para facilitar a edição do código das interfaces e dos objetos que as implementam, é possível adicionar aos nodos de interface objetos do tipo método (explicados na seção 3.4.10). Isso facilita tanto a visualização dos métodos que fazem parte da interface como a edição do código, pois o esqueleto do método é inserido automaticamente.

3.4.4.1 Propriedades

TABELA 3.2 - Propriedade do nodo de interface.

propriedade	descrição
gerador de código	responderá às ações de criação, inserção de métodos e trocas de nomes dos objetos, entre outras

Como mostra a TABELA 3.2, o nodo de interface deve possuir apenas uma propriedade, que é o seu gerador de código. Ele possui essencialmente a mesma estrutura do gerador de código de um nodo básico, sendo mais simples em virtude de que muitas atividades desempenhadas pelos nodos básicos não existem nos nodos de interface.

A existência do nodo de interface foi dúvida durante o projeto da linguagem. Inicialmente pensou-se que uma invocação remota de método pudesse ser representada unicamente pelos objetos cliente e servidor. Analisando-se posteriormente as implementações dos sistemas de objetos distribuídos mais conhecidos (RMI, CORBA, Voyager), verificou-se que em todos era necessária a definição explícita da interface. Isso não seria motivo suficiente para se inserir um novo objeto na linguagem, visto que a ferramenta poderia abstrair essa parte e realizar automaticamente as definições das interfaces. Entretanto, como foi explicado na seção 3.1.1.2, o objetivo desta ferramenta não é impor abstrações ao usuário para que ele tenha uma programação em alto nível, mas sim ser uma ferramenta de apoio à construção de programas que mantenha a flexibilidade das linguagens textuais. A solução dada a este problema foi a inserção do objeto nodo de interface na linguagem visual. Razões relacionadas a essas determinaram a existência dos componentes procuradores e de criação dinâmica e são apresentadas nas seções 3.4.8 e 3.4.9.

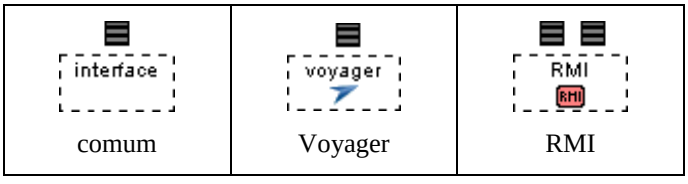


FIGURA 3.14 - Nodos de interface.

Assim como acontece com os nodos básicos, a representação visual dos nodos de interface pode ser configurada pelo usuário. A representação dos nodos implementados neste trabalho segue o modelo que está na FIGURA 3.14, onde é possível se observar também os métodos que podem ser inseridos nos nodos de interface.

3.4.5 Nodo - Nodo virtual

O nodo virtual é assim chamado porque ele não representa um objeto da aplicação que seja implementado pela ferramenta. Ainda que faça parte da aplicação, o nodo virtual apenas indica o acesso da aplicação a um recurso já existente. Esse recurso pode ser qualquer entidade externa, como servidores SMTP ou HTTP, arquivos ou até recursos físicos como impressoras.

Não é gerado código diretamente para os nodos virtuais. Eles fazem parte da aplicação a partir de relacionamentos com outros objetos, que buscam neles os parâmetros de acesso. No exemplo dos servidores SMTP, o nodo que está conectado a ele toma o número da porta e o endereço para conexão com o servidor, que são definidos pelo programador quando se insere um nodo virtual no grafo da aplicação.

Fisicamente, a forma de acesso a um nodo virtual depende da forma de comunicação que ele utiliza. Se esse nodo virtual for, por exemplo, um objeto servidor RMI, o objeto que deseja ter acesso a esse nodo virtual deve utilizar um componente procurador RMI. Se o acesso for através de portas *sockets*, como no caso do servidor SMTP, o componente deve ser uma porta de comunicação comum implementada com *sockets*.

FIGURA 3.15 - Nodos virtuais.

Quanto à representação visual do nodo virtual, obviamente a mais adequada é aquela que indique diretamente o recurso, como uma imagem ou o seu próprio nome. No caso da FIGURA 3.15, há o nodo virtual representando uma máquina que oferece um serviço TCP (contatada por nome e porta), uma impressora, um arquivo, um objeto que oferece métodos implementados com Voyager e outro com RMI.

3.4.6 Nodo - Nodo terminal

Os nodos básicos são objetos que podem ter seu código editado pelo programador durante a construção da aplicação, recebendo suporte da ferramenta para isso. O código é modificado à medida que as suas propriedades são alteradas e componentes são manipulados. Entretanto, em determinados casos isto não é necessário e objetos já desenvolvidos e até compilados podem ser inseridos na aplicação. A utilidade prática mais comum para este tipo de objeto é a reutilização de objetos, como o uso de servidores prontos no grafo da aplicação.

Objetos já prontos são representados nesta linguagem pelos nodos terminais. Eles são formados a partir de nodos básicos e nodos de interface já criados, adicionados de informações que permitam à ferramenta manipulá-los a partir de seus códigos compilados. Na prática, isto significa que a ferramenta, tendo à sua disposição somente o código objeto do nodo básico ou do nodo de interface, deve ser capaz de descobrir

quais objetos fazem parte do nodo, quais são as propriedades desses objetos e assim possa manipulá-los da mesma maneira que o faz com os outros nodos básicos.

FIGURA 3.16 - Nodo terminal.

Como mostra a FIGURA 3.16, o nodo terminal é um objeto que engloba nodos básicos e componentes que foram transformados. Durante a edição dos programas, esses objetos são tratados da mesma maneira que antes da conversão, exceto pelo fato de que seus códigos não podem mais ser alterados.

Deseja-se com este tipo de objeto formar uma biblioteca de componentes para esta ferramenta, de maneira que qualquer nodo básico ou nodo de interface desenvolvido possa ser transformado em nodo terminal e assim ser utilizado posteriormente. Para facilitar a sua utilização, é interessante também que a implementação da ferramenta forneça meios para realizar a conversão de nodos básicos e de interface para nodos terminais.

Como exemplos do uso de nodos terminais, podem ser citados servidores gerais já prontos. Na seção que trata da implementação destes nodos (seção 4.6.6), têm-se os exemplos de um servidor de *lock* e de um servidor de dados globais.

3.4.6.1 Propriedades

As propriedades de um nodo terminal dizem respeito a onde buscar as informações sobre o nodo básico ou nodo de interface correspondente.

TABELA 3.3 - Propriedades de um nodo terminal.

propriedade	descrição
nome da classe ou da interface	nome do nodo básico ou nodo de interface que contém o código compilado correspondente
propriedades do objeto contido	contêm todos os componentes, métodos e outras propriedades do nodo básico ou do nodo de interface

Pelo fato de que o nodo virtual representa um conjunto de objetos, a sua representação visual deve apresentar esse aspecto, como mostra a FIGURA 3.16. Isto permite que tanto o nodo terminal como os componentes individuais possam ser referenciados e participar de relacionamentos.

3.4.7 Nodo - Componente

Componente é um objeto com características próprias e independentes de contexto que lhe dão principalmente capacidade de reutilização e modularidade. Pelo fato de se basearem numa arquitetura padronizada, podem ser desenvolvidos externamente e acoplados ao sistema de forma natural. Nesta linguagem, os objetos componentes são adicionados aos nodos básicos e podem realizar os tipos mais diversos de funções. Podem implementar um simples temporizador ou atividades mais complexas como o controle do acesso a uma base de dados.

O nodo do tipo componente foi incluído nesta linguagem por dois motivos principais: aproveitar componentes já existentes desenvolvidos em outras ferramentas e implementar a comunicação física entre nodos básicos localizados em diferentes máquinas. Para esta última finalidade, existem dois tipos especiais de componentes que serão melhor descritos posteriormente (componente procurador e componente para criação dinâmica). Outros componentes para comunicação disponíveis são as portas de comunicação, que implementam a comunicação por meio de *sockets*.

O componente é definido em termos de propriedades, métodos e eventos. As propriedades podem ter seus valores alterados durante o projeto através da janela de edição de propriedades da ferramenta ou em tempo de execução por meio de código fonte que explicitamente dê valores a elas. Os eventos são responsáveis pela ligação entre o componente e o nodo básico que o contém. O nodo básico possui uma referência ao componente e se registra nele para receber a notificação dos eventos por ele gerados. Além de ser notificado dos eventos, o nodo básico pode executar métodos do componente do mesmo modo que é feito com qualquer outro objeto.

Uma das principais potencialidades de um modelo baseado em componentes é a possibilidade de se estender as capacidades do sistema. Quando uma funcionalidade desejada não estiver disponível, basta implementar um novo componente ou buscar um já pronto e adicioná-lo ao sistema. Esta possibilidade aumenta bastante a flexibilidade do ambiente de programação.

Tendo em vista que novos componentes podem ser adicionados e que suas funções podem não ser conhecidas pela ferramenta, a representação visual de cada componente é implementada por ele próprio. Isto significa que o usuário pode personalizar a apresentação dos componentes, dando-lhes assim a aparência visual que melhor os represente. Por este motivo, um nodo do tipo componente não possui propriedades específicas, como possuem os nodos básicos e outros tipos de nodos. Isso permite que qualquer componente desenvolvido externamente possa ser adicionado a um grafo sem restrição.

Em termos da linguagem visual, há dois tipos de componentes a serem considerados:

- componente sem conexão externa;
- componente com conexão externa.

3.4.7.1 Componente sem conexão externa

Um componente sem conexão externa é um componente utilizado por um objeto que realiza determinadas funções que não possuem nenhum relacionamento explícito com outros objetos. Exemplos desse tipo de componente podem ser um temporizador, que gera eventos segundo uma determinada freqüência e um componente de interface gráfica. Como eles não precisam apresentar conexão externa, as suas representações visuais são simplesmente os seus ícones que acompanham o nodo básico, como está na FIGURA 3.17.

FIGURA 3.17 - Componentes sem conexão externa ligados a um nodo básico.

Este tipo de componente é o mais comum, pois não necessita de nenhuma informação adicional para ser usado. Componentes desenvolvidos por terceiros para outras ferramentas são automaticamente considerados componentes sem conexão externa.

3.4.7.2 Componente com conexão externa

Como foi visto na seção anterior, existem componentes que não necessitam de nenhum tipo de conexão externa. Entretanto, em outros isto é necessário para que haja a visualização de um relacionamento, como uma invocação remota de método ou criação remota de objeto. A representação visual do relacionamento nesses casos é uma necessidade para se melhor compreender o funcionamento da aplicação neste ponto.

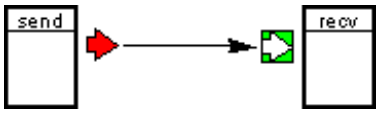


FIGURA 3.18 - Componentes com conexão externa.

A FIGURA 3.18 mostra uma porta de saída de dados conectada a uma porta *socket* de entrada de dados. A existência do arco entre os dois facilita bastante a compreensão de como os nodos básicos *send* e *recv* se relacionam.

Por causa da possibilidade de estabelecer relacionamentos, o componente com conexão externa deve prover informação sobre como serão esses relacionamentos. A forma de como isso é feito será explicada com detalhes na seção 3.4.14.

3.4.8 Nodo - Componente procurador

O componente procurador é uma referência virtual local a um objeto remoto. É este tipo de objeto que faz com que a aplicação utilize todo o potencial dos objetos distribuídos e assim realize invocação remota de métodos de forma transparente. O objeto local que precisa ter acesso a métodos remotos o faz através do componente procurador, que após ser iniciado com as informações do objeto remoto é capaz de oferecer uma interface de acesso a ele.

A existência desse tipo de objeto é uma conseqüência da implementação dos principais *middlewares* de objetos distribuídos, como RMI e Voyager. Nos dois sistemas, a chamada de métodos remotos é feita através de procuradores locais, obtidos a partir de uma consulta ao servidor de nomes, os quais retornam uma referência para o objeto remoto.

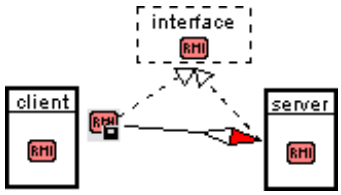


FIGURA 3.19 - Componente procurador, conectado à interface e ao objeto remoto.

A FIGURA 3.19 mostra como é a expressão de uma invocação de método remoto utilizando-se um componente procurador (o exemplo mostra a implementação com RMI, mas a representação para Voyager e outros sistemas é análoga). Percebe-se que é necessário um número relativamente grande de elementos visuais para completar a representação (sete ao todo - três nodos, um componente e três arcos). À primeira vista pode parecer que há funções repetidas nesses objetos, considerando-se que numa

invocação remota de método, numa abstração de alto nível, existem apenas o objeto que chama e o que executa a chamada (FIGURA 3.20).

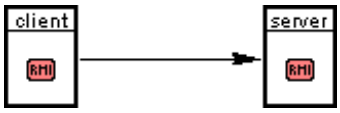


FIGURA 3.20 - Abstração em alto nível de uma invocação de método.

Entretanto, este tipo de representação mais simples não é capaz de expressar todas as situações que podem ocorrer. Nestes relacionamentos, as seguintes situações são exemplos de onde esta visualização não se aplicaria, visto que uma mesma representação seria empregada para todas elas:

- o objeto local faz acesso a duas interfaces implementadas por um objeto remoto. Isto exige, portanto, a presença do objeto procurador que faz a diferenciação entre as duas interfaces;
- um objeto distribuído remoto implementa diferentes interfaces: o componente procurador deve, portanto, especificar qual a interface que vai ser usada. Por isso, não basta que ele se relacione apenas com o objeto remoto, mas que se relacione também com o nodo de interface;
- uma mesma interface pode ser implementada por diferentes objetos remotos: o componente procurador deve portanto especificar qual o objeto distribuído que será chamado. Assim, não pode se relacionar apenas com a interface.

As três situações descritas acima demonstram que é necessária a presença desses três tipos de objetos na representação de uma chamada de método remoto. Sem um deles, a linguagem não é poderosa o suficiente para expressar com clareza todas as construções possíveis dentro deste modelo de programação.

Em vista da impossibilidade de simplificação da linguagem sem perdas na capacidade de expressão, algumas medidas podem ser tomadas com o fim de diminuir a complexidade da apresentação visual de uma invocação remota. Numa delas, a implementação poderia prover meios para que a exibição de determinados objetos fosse opcional, diminuindo assim o número de objetos visualizados. Por exemplo, caso cada objeto remoto implementasse apenas uma interface (caso mais comum), ela poderia ser omitida na visualização, embora ainda fizesse parte do programa. Essa opção ficaria a critério do usuário.

3.4.8.1 Propriedades

TABELA 3.4 - Propriedades de um componente procurador.

propriedade	descrição
nome remoto	nome com que o objeto remoto foi registrado no servidor de nomes
nodo pai	nodo básico que criou este objeto
caminho do grafo	identificação do nível atual de instanciação do grafo
servidor de nomes	endereço do servidor de nomes que armazena os dados dos objetos da aplicação distribuída
retorna objeto remoto	método que retorna uma referência local ao objeto remoto

Um componente procurador deve apresentar as propriedades da TABELA 3.4 para que consiga desempenhar todas as suas funções. A principal delas é o método que retorna a referência local ao objeto remoto. Para realizar esta tarefa, o nodo procurador precisa conhecer o endereço do servidor de nomes e com que nome o objeto remoto foi nele registrado.

3.4.9 Nodo - Componente para criação dinâmica

A função principal deste tipo de componente é permitir a instanciação dinâmica de objetos distribuídos. Um nodo básico que contenha um componente para criação dinâmica está apto a criar objetos tanto localmente como remotamente. Para isso, deve haver também um relacionamento entre esse componente e o objeto remoto para que sejam adquiridas as informações necessárias à instanciação.

A necessidade deste tipo de objeto deve-se à incapacidade do sistema de definir com certeza o exato momento em que a instanciação do objeto remoto é necessária. Por isso, esse componente fica disponível ao usuário para que ele informe explicitamente, no código do objeto criador, o instante da criação do objeto. A outra opção seria liberar o usuário dessa tarefa e deixá-la para a ferramenta. Entretanto, quando a criação dos objetos é condicional ou não deve ser feita logo no início da aplicação (caso este em que os objetos são criados automaticamente), a ferramenta não tem condições de obter por si só as informações necessárias para esse trabalho.

Algumas ferramentas visuais são capazes de realizar criação de objetos em momentos determinados na execução da aplicação. Entretanto, quando isso não é feito explicitamente pelo usuário, é utilizado algum tipo de anotação especial com o uso de expressões que devem ser testadas a intervalos regulares segundo um modelo particular de programação. Como já foi falado, a existência de um modelo particular de programação concorrente, que exige muitas vezes uma linguagem especial, foi descartada aqui em favor da flexibilidade que seria perdida com essa solução.

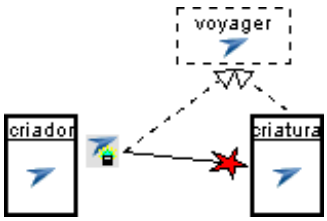


FIGURA 3.21 - Componente para criação dinâmica.

O componente de criação dinâmica retorna automaticamente uma referência local ao objeto remoto, passando a se comportar como um componente procurador depois que o objeto foi criado. Assim, não é necessário que haja um componente procurador adicional para realizar as chamadas de métodos remotos. Esse é o motivo pelo qual o componente de criação dinâmica necessita se relacionar também com o nodo de interface. Como mostra a FIGURA 3.21, ele é acoplado ao nodo básico criador e se relaciona com o objeto e a interface remotos.

3.4.9.1 Propriedades

A propriedade que caracteriza este tipo de componente é o método que cria um objeto remoto. Ele usa as informações das outras propriedades (máquina remota, endereço do servidor de nomes, classe do objeto que será criado) para instanciar o objeto remoto, retornando uma referência local para ele, do mesmo modo que é feito pelos componentes procuradores.

TABELA 3.5 - Propriedades de um componente para criação dinâmica.

propriedade	descrição
máquina	nome da máquina onde será criado o objeto remoto
nome da classe remota	classe do objeto que será criado remotamente
nome remoto	nome com que o objeto remoto será conhecido na rede
nodo pai	nodo básico que criou este componente
caminho do grafo	identificação do nível de instanciação em que o grafo atual se encontra
servidor de nomes	endereço do servidor de nomes
cria objeto remoto	método que cria um objeto remotamente e retorna uma referência local ao objeto criado

Os objetos remotos criados são sempre nodos básicos. Após a criação, o método *run* deve ser chamado no nodo básico para que ele inicie a sua execução.

3.4.10 Nodo - Método

O objeto do tipo método é inserido no grafo da aplicação com a função de facilitar a visualização gráfica dos métodos de uma interface ou de uma classe. A visualização é feita tanto no grafo como na janela de projeto.

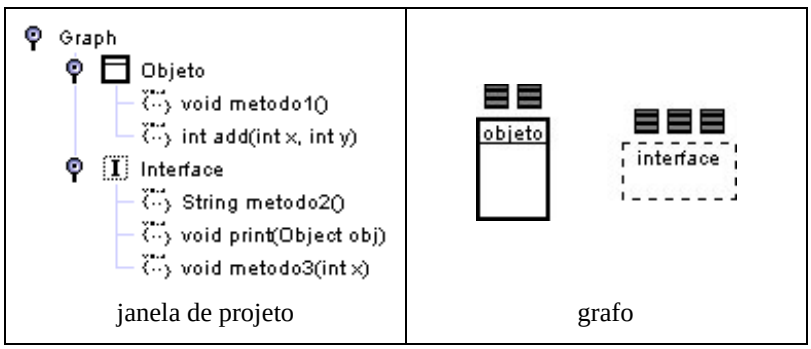


FIGURA 3.22 - Métodos em nodos básicos e interfaces.

A FIGURA 3.22 mostra como o objeto método ajuda a visualizar os métodos existentes na interface, indicando portanto quais os métodos que o nodo básico conectado a esta interface deve implementar. Como será visto no capítulo de implementação, este tipo de objeto facilita também a construção dos programas, pois quando um método é inserido na interface, ao código de todos os nodos básicos que implementam essa interface é adicionado o esqueleto da implementação do método.

3.4.11 Nodo - Nodo de interface de grafo

Antes de se disponibilizar um grafo para que seja empregado em outros programas, sua interface de acesso deve ser fornecida. Isto é feito através dos nodos de interface de grafo, que indicam num grafo quais objetos serão visíveis externamente numa chamada a grafo. No grafo da FIGURA 3.23, o nodo básico *obj1* e o nodo de interface *interface* estarão disponíveis numa chamada a esse grafo.

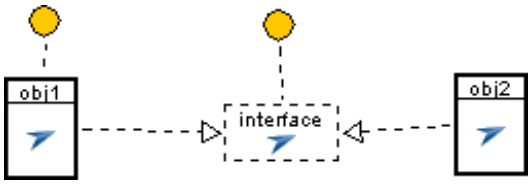


FIGURA 3.23 - Interfaces de acesso ao grafo.

Um grafo pode ser composto por vários objetos. Os objetos que são referenciados externamente devem ser conectados a uma interface de acesso para que sejam visíveis numa chamada a grafo. Os nodos de interface de grafo não têm nenhuma função dentro do programa do usuário e também não provocam alterações no código dos nodos quando inseridos no grafo. Sua única função é informar qual nodo estará acessível externamente. Quando esse grafo for inserido em outro através de um nodo de chamada a grafo, somente os objetos conectados aos nodos de interface de grafo estarão disponíveis.

3.4.12 Nodo - Chamada a grafo

Chamada a grafo é o objeto da linguagem que representa visualmente um outro grafo (que neste contexto pode também ser chamado de subgrafo). Grande parte das linguagens de programação visual suporta a definição de subgrafos, procurando atingir principalmente os seguintes objetivos:

- **reutilização de código**: um grafo é salvo numa biblioteca, onde posteriormente pode ser buscado e utilizado em outros pontos do programa ou até mesmo em outros programas;
- **modularização da aplicação**: a definição modular e hierárquica da aplicação facilita o desenvolvimento de projetos, principalmente os de maior porte;
- **escalabilidade da linguagem visual**: um dos grandes problemas das linguagens visuais é a sua escalabilidade [BUR95]. A definição hierárquica em termos de subgrafos diminui o tamanho visual dos programas, facilitando a sua visualização.

Para que o grafo possa ser utilizado é preciso que se definam as suas interfaces. Como visto na seção anterior, isso é feito com a ajuda dos nodos de interface de grafo. Quando um nodo de chamada a grafo é inserido num grafo, os objetos que estão conectados às interfaces são exibidos juntamente com o nodo de chamada, permitindo que eles façam parte da aplicação como quaisquer outros objetos. A FIGURA 3.24 exibe o nodo de chamada ao grafo da FIGURA 3.23, que contém na sua interface um nodo básico e um nodo de interface.

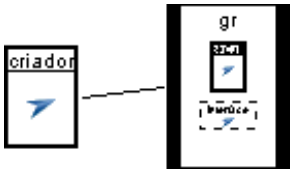


FIGURA 3.24 - Chamada a grafo.

O grafo a ser chamado pode ser qualquer grafo já construído e assim conter vários nodos básicos e nodos terminais. O grafo chamado pode ser inclusive ele mesmo, o que possibilita portanto a criação recursiva de grafos. Isto é muito útil na implementação de determinados algoritmos, como o *quicksort* (vide representação visual na seção 4.11.4), onde a definição recursiva é o meio mais natural de se descrever a solução do problema.

O momento de criação de um subgrafo é indefinido. Ele pode tanto ser criado de forma automática, simultaneamente aos outros nodos básicos e nodos terminais do grafo, bem como ser criado pelo programador através de um nodo básico conectado a ele (no exemplo da FIGURA 3.24, a conexão entre o nodo básico *criador* e o nodo de chamada a grafo indica que o primeiro é responsável pela criação do segundo). Os fatos de que nem sempre é necessária a criação do subgrafo, de que o instante da necessidade dos serviços do grafo é indefinido e de que é possível se fazer chamadas recursivas a grafos são os motivos pelos quais a criação do grafo deve ser explicitada pelo programador em certos casos.

3.4.12.1 Propriedades

TABELA 3.6 - Propriedade de um nodo de chamada a grafo.

propriedade	descrição
nome do grafo	nome do grafo que será chamado

A única propriedade necessária a um nodo de chamada a grafo é o nome do grafo a ser chamado. Na implementação do nodo de chamada a grafo feita neste trabalho (vide seção 4.8), o grafo é identificado por um nome de arquivo.

3.4.13 Nodo - Documentação

Este tipo de nodo não possui nenhuma influência na execução do programa. Ele é utilizado apenas para a documentação e apresentação visual do programa ao usuário.

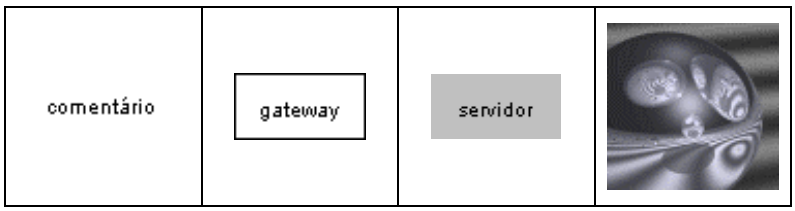


FIGURA 3.25 - Nodos de documentação.

Como mostra a FIGURA 3.25FIGURA 3.25 - , os nodos de documentação podem ser imagens, campos textuais e quaisquer outros tipos de representação usados com fins de apresentação. Fazendo-se uma analogia com as linguagens de programação textual, o nodo de documentação é o "comentário" desta linguagem.

3.4.14 Arco

O arco é o elemento da linguagem que explicita graficamente os relacionamentos entre os diferentes objetos da aplicação. É ele que faz a diferença entre a programação visual e a programação textual, tornando a expressão de relacionamentos e ligações mais significativa e por isso importante em certos contextos, tais como a programação concorrente. Na programação textual não existe uma entidade desse tipo que se destina essencialmente a expressar relacionamentos e ligações. É por isso que relacionamentos como trocas de mensagens e chamadas remotas de método são difíceis de se perceber nesses programas. Para um ser humano, a representação visual de tais interações é consideravelmente mais expressiva do que a representação textual da mesma situação.

A semântica do arco varia conforme os tipos de objetos que estão sendo relacionados. Assim, combinando todos os tipos de objetos da linguagem, obtêm-se os relacionamentos da TABELA 3.7, cujas identificações estão na TABELA 3.8.

TABELA 3.7 - Relacionamentos.

Fonte \ Destino	Nodo básico	Nodo de interface	Nodo virtual	Nodo terminal	Componente	Procurador	Criação dinâmica	Método	Interface de grafo	Chamada a grafo	Documentação
Nodo básico	3	2	1	-	1	1	1	1	8	7	9
Nodo de interface	-	-	-	-	-	-	-	-	8	-	9
Nodo virtual	-	-	-	-	-	-	-	-	8	-	9
Nodo terminal	-	-	-	-	-	-	-	-	8	-	9
Componente	1	1	1	-	1	1	1	1	8	-	9
Procurador	4	6	1	-	1	1	1	1	8	-	9
Criação dinâmica	5	6	1	-	1	1	1	1	8	-	9
Método	-	-	-	-	-	-	-	-	-	-	9
Interface de grafo	8	8	8	8	8	8	8	-	-	-	9
Chamada a grafo	-	-	-	-	-	-	-	-	-	-	9
Documentação	9	9	9	9	9	9	9	9	9	9	9

TABELA 3.8 - Tipos de relacionamentos.

Identificador	**Relacionamento**
1	normal
2	implementação de interface
3	compartilhamento de memória
4	referência a objeto remoto
5	criação de objeto remoto
6	interface de objeto remoto
7	criação de grafo
8	interface de grafo
9	documentação

A coluna da esquerda da TABELA 3.7 indica o nodo fonte do relacionamento e a linha de cima, o destino. Isso significa que, por exemplo, um arco partindo de um nodo básico e chegando num nodo de interface estabelece o relacionamento 2, ou seja, uma implementação de interface. Os cruzamentos preenchidos com um traço indicam que não existe importância no significado da interação entre os objetos correspondentes.

A tabela produziu ao todo nove tipos de relacionamentos:

- **normal**: é um relacionamento não especial, onde há apenas amarração entre propriedades dos objetos fonte e destino;
- **implementação de interface**: o nodo origem implementa a interface representada pelo nodo destino;

- **compartilhamento de memória**: os objetos fonte e destino compartilham o mesmo espaço de endereçamento;
- **referência a objeto remoto**: a origem do arco está chamando um método remoto que é executado no objeto destino do arco;
- **criação de objeto remoto**: o nodo fonte do arco cria o objeto destino dinamicamente;
- **interface de objeto remoto**: o objeto fonte do relacionamento conhece a interface implementada pelo objeto remoto;
- **criação de grafo**: o nodo origem cria o grafo destino dinamicamente;
- **interface de grafo**: o objeto conectado ao nodo de interface de grafo será visível externamente ao grafo;
- **documentação**: indica apenas a que objeto se refere determinada documentação.

A FIGURA 3.26 mostra uma representação para os diferentes tipos de arcos. A linguagem visual não define como deve ser a aparência do arco, mas a implementação deve fazer com que os diferentes tipos de relacionamentos sejam representados visualmente de maneira diversa entre si e com a forma apropriada ao caso a que se referem. Isto permite flexibilidade na visualização e também que novos relacionamentos adicionados pelo usuário sejam configurados conforme a sua vontade.

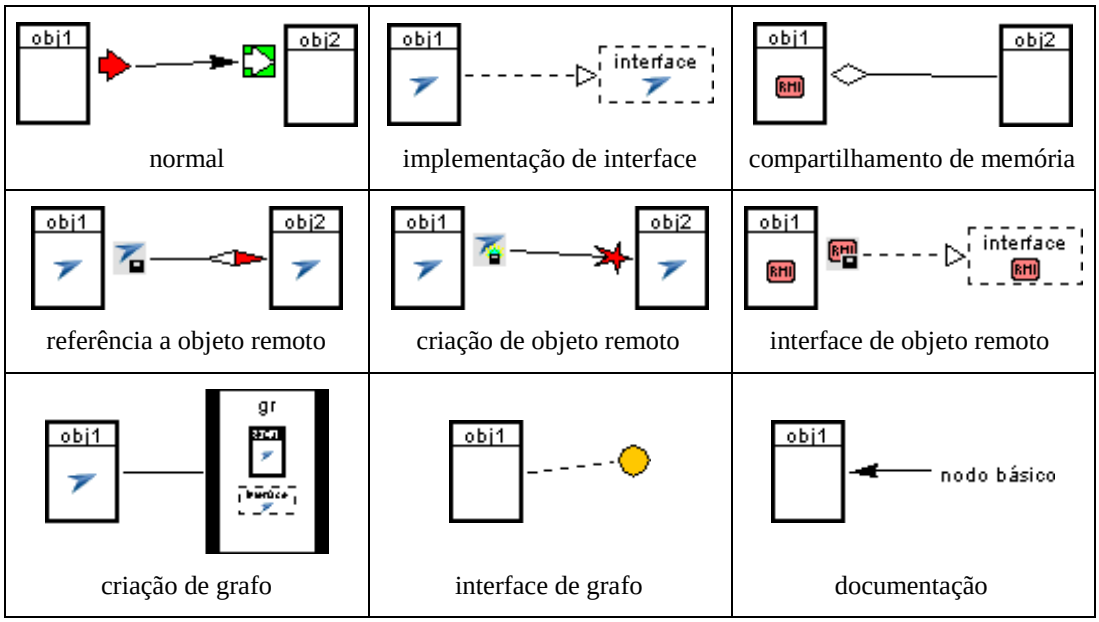


FIGURA 3.26 - Diferentes tipos de arcos.

As seguintes seções explicarão com mais detalhes o significado de cada um desses tipos de relacionamentos.

3.4.15 Arco - Normal

Quando os objetos relacionados não possuem nenhum tipo de relacionamento especial entre si (invocação remota ou implementação de interface, por exemplo), a semântica da relação entre esses objetos varia conforme o caso. As seções 3.4.15.1, 3.4.15.2 e 3.4.15.3 explicam três desses casos.

O relacionamento é concretizado através da amarração de propriedades entre o nodo fonte e o nodo destino do arco. Quando o programador edita o grafo e insere um arco, ele deve definir quais propriedades se ligarão entre a fonte e o destino. Isso faz com que essas propriedades tenham sempre o mesmo valor em tempo de desenvolvimento, o que na prática implementa o relacionamento. Esse tipo de amarração funciona de maneira semelhante às chaves que relacionam duas tabelas de um banco de dados (que neste caso, inclusive, também é chamado de relacionamento).

Para exemplificar, serão apresentados os casos do relacionamento de transferência de dados por portas de comunicação, de uma aplicação cliente-servidor e de acesso a um recurso.

3.4.15.1 Transferência de dados

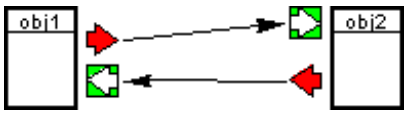


FIGURA 3.27 - Relacionamentos de transferência de dados.

Um exemplo de relacionamento representado por um arco normal, onde há somente amarração de propriedades, ocorre entre uma porta de comunicação de saída e uma porta de comunicação de entrada (no exemplo da FIGURA 3.27, existem dois relacionamentos desse tipo). Essas duas portas são implementadas com *sockets*, fazendo com que a porta de saída deva conhecer o nome da máquina e o número da porta remota.

Essa necessidade de que um objeto deva conhecer os valores das propriedades de outros objetos é o indício de que deve haver amarração de propriedades. A TABELA 3.9 mostra como ela acontece nesse exemplo.

TABELA 3.9 - Amarração de propriedades entre as portas de comunicação.

Porta de saída	Porta de entrada
cor da frente	cor da frente
cor de fundo	cor de fundo
número da porta remota	número da porta local
nome da máquina remota	nome da máquina local
identificador do protocolo	identificador do protocolo

Os dois objetos apresentam inicialmente diversas propriedades. Algumas podem ser indispensáveis para o funcionamento do componente e outras podem ser utilizadas apenas para a apresentação visual no editor de grafos. Quando se cria o relacionamento, todas elas estão à disposição do usuário. Para que a criação do relacionamento seja efetivada, o usuário deve informar quais propriedades do objeto fonte se combinam com quais propriedades do objeto destino. Não é necessário que todas tenham um correspondente no outro lado e nem que duas propriedades com o mesmo nome tenham que se combinar. As únicas restrições são as de que um mapeamento envolva sempre duas propriedades e que essas propriedades sejam do mesmo tipo de dado. Um relacionamento será formado portanto por um ou mais mapeamentos, como mostra o exemplo da TABELA 3.9, que expressa o relacionamento da FIGURA 3.27.

Percebe-se que há diversos passos na construção de um relacionamento desse tipo. Além disso, muitas vezes o usuário pode não ter o conhecimento do significado das propriedades, o que o impede de criar os relacionamentos com a certeza de que o processo foi feito corretamente. Para resolver esses problemas, o desenvolvedor da ferramenta pode oferecer algumas facilidades. Como é explicado na seção 4.7.1 que trata da implementação do arco normal, os componentes podem conter informações que automatizem a criação destes tipos de relacionamentos.

3.4.15.2 Aplicação cliente-servidor

Um dos tipos de aplicações que essa ferramenta se propõe a priorizar são as aplicações cliente-servidor. Elas podem ser implementadas de diversas maneiras, desde *sockets* até chamadas de métodos remotos.

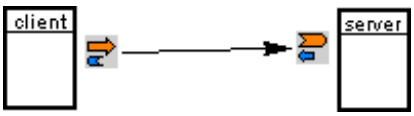


FIGURA 3.28 - Aplicação cliente-sevidor com portas *sockets*.

Nesta ferramenta, a representação visual da implementação com *sockets* de uma aplicação cliente-servidor é feita da maneira mostrada na FIGURA 3.28. Neste programa, o objeto *client* utiliza uma porta de chamada de serviço para se conectar ao objeto *server*, que por sua vez tem uma porta de recebimento de conexões que fica continuamente esperando por pedidos de serviço. Quando uma nova requisição chega, essa porta cria um novo objeto para tratar a conexão criada com o cliente recém conectado. Cada objeto que trata clientes possui uma porta que se comunica diretamente com eles, sem passar pela porta de conexão, a qual volta a cuidar das requisições que continuam chegando.

TABELA 3.10 - Amarração de propriedades entre uma porta de comunicação e um nodo virtual.

chamada de serviço	**conexão de serviço**
cor da frente	cor da frente
cor de fundo	cor de fundo
	nome do serviço
número da porta remota	número da porta local
nome da máquina remota	nome da máquina

Para que a porta de chamada de serviço conheça os parâmetros do servidor, deve-se criar o relacionamento entre essa porta e a porta de conexão de serviço. Assim como no caso da transferência simples de dados, devem ser conhecidos o número da porta e o endereço do servidor remoto. Na porta de conexão, deve ainda ser especificado o nome do serviço, o qual irá determinar quais objetos serão criados quando as novas requisições chegarem.

3.4.15.3 Acesso a um recurso

Outro exemplo de arco normal é o relacionamento que indica o acesso da aplicação a um recurso externo, representado no grafo por um nodo virtual. É através deste arco que um objeto conhece os parâmetros de acesso ao recurso.

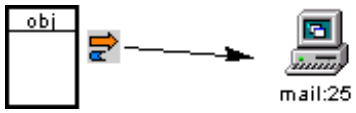


FIGURA 3.29 - Arco indicando acesso a um recurso externo.

Por exemplo, na FIGURA 3.29, o acesso ao servidor virtual (um servidor de correio eletrônico disponível na máquina *mail* na porta 25) é feito através de uma porta de chamada de serviço. As propriedades dessa porta e do nodo virtual, bem como a combinação entre elas é mostrada na TABELA 3.11.

TABELA 3.11 - Amarração de propriedades entre uma porta de comunicação e um nodo virtual.

chamada de serviço	nodo virtual
cor da frente	cor da frente
cor de fundo	cor de fundo
número da porta remota	número da porta local
nome da máquina remota	nome da máquina

As propriedades das cores não são utilizadas no relacionamento. Cada objeto possui essas propriedades com a finalidade única de apresentação visual no grafo.

3.4.16 Arco - Implementação de interface

É um relacionamento definido entre um nodo básico e um nodo de interface, indicando que o objeto representado pelo primeiro implementa a interface correspondente ao segundo, como mostra a FIGURA 3.30. Nesse exemplo, o objeto *obj1* implementa as interfaces *interface1* e *interface2*.

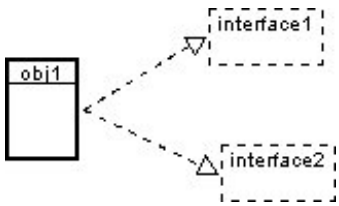


FIGURA 3.30 - Arcos de interface.

Este relacionamento é utilizado principalmente na definição de interfaces de objetos remotos. É através dessas interfaces que os objetos clientes adquirem o conhecimento necessário a respeito de como fazer o acesso remoto. Outra finalidade para esta interação é a implementação simples de uma interface por um objeto, como acontece nos programas seqüenciais feitos em Java.

Na implementação, a representação visual para este arco é a mesma utilizada na linguagem de modelagem UML [FOW97][OMG2000a] para a implementação de interface. Assim como acontece com os nodos básicos e de interface, a representação desses arcos também pode ser personalizada.

3.4.17 Arco - Memória compartilhada

Na representação normal de nodos no grafo, cada nodo possui o seu próprio espaço de endereçamento. Embora fisicamente possam até estar na mesma máquina, dois nodos não têm condições de compartilhar dados diretamente. Caso necessária, a comunicação entre os dois deve se dar através de outros meios. Essa é uma restrição

indesejável em certos problemas (principalmente quando dois objetos devem se comunicar com grande freqüência) e por isso o modelo foi flexibilizado a fim de que dois objetos possam então ter acesso ao mesmo espaço de memória. Na prática, isto significa que os objetos podem fazer chamadas de método diretamente entre si.

Como mostrou a tabela que especifica todos os relacionamentos (TABELA 3.7), o compartilhamento de memória ocorre somente entre nodos básicos. Entretanto, o nodo básico fonte do arco deve ser obrigatoriamente um nodo básico normal, isto é, não pode ser parte de um nodo terminal ou de uma chamada a grafo. Esta restrição existe porque o código fonte precisa ser alterado quando um arco desse tipo é criado.

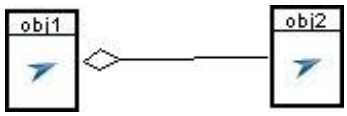


FIGURA 3.31 - Arco representando compartilhamento de memória entre nodos básicos.

A representação visual para este tipo de arco foi inspirada na representação de objetos compostos em diagramas de estruturas estáticas em UML, já que a sua implementação (vide seção 4.7.3) envolve composição de objetos.

Além de permitir compartilhamento de memória, outra importante utilidade deste tipo de relacionamento é a de possibilitar que um objeto pertencente a um nodo básico possa ter seu código alterado. Em situações normais, o caso mais comum é o de adicionar um componente a um nodo básico. Porém, o código dos componentes não pode ser alterado, pois eles são incluídos já em formato de código objeto. Adicionar um nodo básico a outro é o mesmo que adicionar um componente, exceto que num nodo básico é possível alterar o seu código e, inclusive, adicionar ainda outros componentes e nodos básicos.

3.4.18 Arco - Interface de objeto remoto

Antes que um objeto tenha acesso a métodos remotos, é necessário que ele conheça a interface implementada remotamente. Considerando que nesta linguagem os objetos remotos são referenciados por componentes procuradores e de criação dinâmica, a estes objetos devem ser conectadas as interfaces que serão referenciadas.

FIGURA 3.32 - Arco indicando a interface remota referenciada.

Conhecendo a interface remota, os objetos conectados a ela podem se comportar como se implementassem essa interface. Pelo menos é dessa forma que eles são vistos pelos nodos básicos. No caso da FIGURA 3.32, o nodo básico *obj* trata o componente procurador (objeto que na figura está ao lado de *obj*) como um objeto do tipo definido pela interface *interface*. Desse modo, o nodo básico pode fazer chamadas de métodos nesse procurador como se ele fosse um objeto do tipo *interface* (exemplos mais completos já foram apresentados nas seções 3.4.8 e 3.4.9).

3.4.19 Arco - Referência a objeto remoto

A invocação remota de método é uma das características mais marcantes de um sistema de objetos distribuídos. Ter a possibilidade de realizar chamadas remotas de forma quase transparente e muito semelhante às invocações locais é um recurso

poderoso que facilita bastante o desenvolvimento das aplicações. Portanto, como os objetos distribuídos possuem papel central nessa ferramenta, essa característica não poderia deixar de estar presente.

O relacionamento que representa a invocação remota de método existe entre um componente procurador e um objeto distribuído. O arco, como visto na FIGURA 3.33, parte do componente procurador e mostra ao usuário que o nodo básico de origem *obj1* faz chamadas a métodos remotos que são executados no nodo básico destino *obj2*.

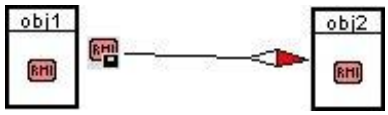


FIGURA 3.33 - Arco indicando invocação remota de método.

Assim como acontece no relacionamento normal entre componentes, neste caso também deve haver a amarração de propriedades. Conforme a implementação, diferentes informações devem ser trocadas entre o nodo procurador e o nodo básico servidor, como o nome segundo o qual o servidor é conhecido na rede e o nome da máquina em que se localiza.

3.4.20 Arco - Criação de objeto remoto

Assim como os grafos, nem todos os objetos são criados logo no início da aplicação. Para os objetos que estiverem nesse caso, meios devem ser oferecidos para que eles sejam criados dinamicamente. Como a ferramenta não tem condições de descobrir por si só o momento em que os objetos devam ser criados, o usuário deve se responsabilizar por essa tarefa. Para isso existe o relacionamento de criação de objeto remoto, que conecta o componente de criação dinâmica ao objeto que será criado.

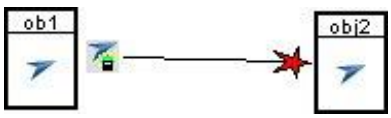


FIGURA 3.34 - Arco representando criação dinâmica de um objeto distribuído.

Este tipo de relacionamento é formado por um componente de criação dinâmica e um nodo básico. Ao se conectar esses dois objetos, o nodo básico que contém o componente possuirá a capacidade de criar objetos remotos da classe do objeto destino. O tipo de relacionamento é bastante semelhante ao de invocação remota de método. Aliás, depois de criado o objeto, o componente de criação dinâmica possui uma referência para o objeto remoto, passando a se comportar como um componente procurador.

3.4.21 Arco - Criação de grafo

Os grafos são objetos que, a menos de uma especificação inicial, não são criados automaticamente no início da aplicação. A criação automática de todos os grafos poderia criar um número excessivo de objetos e, inclusive, iniciar um laço infinito de criação, visto que a linguagem visual permite definição recursiva de subgrafos.

Em vista disso, a responsabilidade da escolha do instante em que será criado o grafo e de qual objeto fará isso é do programador. O objeto que instanciará os objetos do grafo é definido através do arco de criação do grafo. O objeto fonte do arco, que deve ser um nodo básico, possuirá os métodos necessários que serão chamados quando houver a necessidade de criação do grafo.

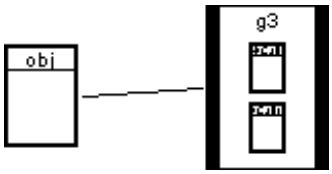


FIGURA 3.35 - Arco indicando que o nodo básico irá criar o grafo.

O exemplo da FIGURA 3.35 mostra que o nodo básico *obj* possuirá o código necessário para a criação do grafo. Na implementação, quando o arco é criado, esse código que cria o grafo é inserido no código do nodo básico.

A cada grafo é permitido somente um arco de criação de grafo. Caso um grafo deva ser instanciado mais de uma vez, novos nodos de chamadas a grafo referentes ao mesmo tipo de grafo devem ser inseridos no grafo. Esta restrição existe porque as identificações dos objetos na rede são únicas para cada grafo. Se um grafo fosse criado duas vezes, dois objetos teriam a mesma identificação (na implementação, provavelmente ocorreria um erro, pois em geral não é possível se atribuir o mesmo nome a dois objetos). Isto vale também para nodos básicos, ou seja, no máximo um arco de criação dinâmica pode estar conectado a um objeto.

3.4.22 Arco - Interface de grafo

Um grafo pode ter um sem número de objetos, mas nem todos eles serão necessariamente referenciados em outros grafos. A fim de evitar que uma interface seja formada por muitos objetos e deixá-la somente com os estritamente necessários, deve-se conectar estes objetos aos nodos de interface de grafo. Neste instante então se cria um relacionamento que indica que tal objeto faz parte da interface do grafo.

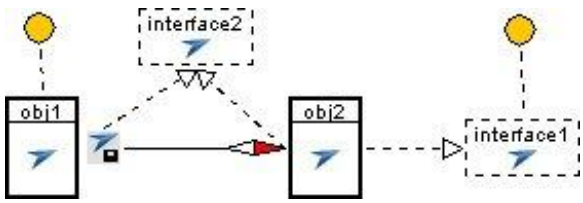


FIGURA 3.36 - Arcos indicando que os objetos são a interface do grafo.

No caso do exemplo do grafo da FIGURA 3.36, quando ele for referido num nodo de chamada a grafo, os objetos *obj1* e *interface1* serão parte da interface do grafo, fazendo com que somente estes objetos sejam visíveis num nodo de chamada a grafo.

3.4.23 Arco - Documentação

O arco de documentação possui a mesma função do nodo de documentação, isto é, apresentar informações ao usuário a respeito do programa dentro do próprio programa, como se fosse um comentário de uma linguagem de programação textual. A sua função é apenas de apresentação e por isso não influi na implementação da aplicação.

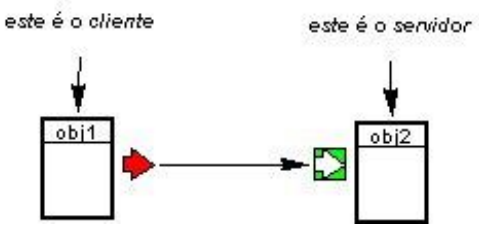


FIGURA 3.37 - Arcos de documentação.

Como está na FIGURA 3.37, os arcos entre os nodos de documentação e os nodos básicos servem para facilitar a função da documentação, principalmente quando há muitos objetos e não se sabe a qual deles se refere um determinado comentário de texto ou imagem.

3.4.24 Grafos

O modelo de grafos é uma parte importante deste modelo. Cada aplicação é composta por pelo menos um grafo que serve de ponto de entrada para a sua execução. Através do nodo de chamada a grafo é possível se construir a aplicação de forma hierárquica, facilitando a visualização e a compreensão da estrutura geral do programa.

Os grafos devem apresentar propriedades que os associem ao ambiente de execução que será utilizado. Dessa forma, para cada ambiente de execução suportado deve ser oferecido um tipo de grafo. Somente assim será possível para a ferramenta gerar o código para a criação dos grafos, pois a instanciação remota de objetos e o registro no servidor de nomes, entre outras atividades, são dependentes da plataforma de execução utilizada.

A FIGURA 3.38 mostra a estrutura de uma aplicação exemplo e como os seus objetos se relacionam. Na parte da esquerda da figura está a estrutura do projeto da aplicação e na da direita a forma como os objetos se distribuem durante a execução. No projeto existem três classes de grafo: grafo 1, grafo 2 e grafo 3, formados por nodos básicos (n1-n7) e pelos nodos de chamada a grafo c1, c2 e c3 que chamam, respectivamente, os grafos 2, 3 e 1.

Os grafos são criados de cima para baixo e referenciados por quem o criou através do nodo de chamada a grafo. Durante a execução, cada objeto recebe um nome que o identifica na aplicação (este nome varia conforme o nível de instanciação do grafo do qual ele faz parte, como está na parte direita da figura).

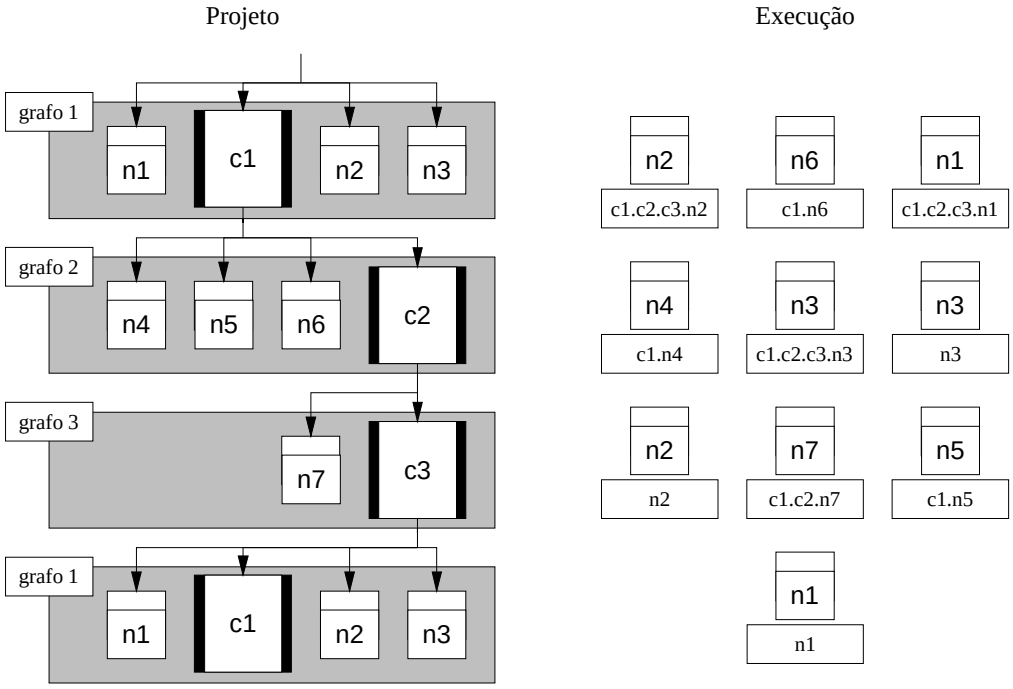


FIGURA 3.38 - Estruturação hierárquica da aplicação.

Durante a execução, todos os objetos são vistos no sistema como sendo objetos iguais, independentemente do nível em que estejam no grafo do projeto (por isso eles são mostrados na figura sem ordem nenhuma). Eles são diferenciados em nível de aplicação através de um identificador que funciona como um caminho de grafo, formado por todos os nodos de chamada a grafo que estão entre a raiz e o objeto. Esse tipo de identificação permite que haja vários grafos do mesmo tipo instanciados ao mesmo tempo, inclusive de forma recursiva. Na figura, por exemplo, há dois nodos da classe n1, referenciados pelos nomes n1 e c1.c2.c3.n1. O nome n1 vem do fato de que o objeto correspondente foi criado a partir do nível mais alto, enquanto o nome c1.c2.c3.n1 indica que ele está no quarto nível, após três chamadas de grafo.

3.4.25 Expressão visual da concorrência

O objetivo desta seção é explicar brevemente como se dá o mapeamento entre os conceitos abstratos de programação concorrente (distribuição e comunicação) para a linguagem visual nesta ferramenta. Isto é importante para que se entenda com maior clareza os tipos de programas que podem ser feitos aqui.

3.4.25.1 Distribuição

Assim como acontece na maioria das linguagens visuais que especificam os programas concorrentes através de grafos, a distribuição e o paralelismo são expressos através do elemento nodo (nesta linguagem, mais especificamente o nodo básico). A separação da aplicação em diferentes fluxos de execução indica que eles estão localizados em espaços de endereçamento disjuntos. Na prática, isso significa que eles podem ser objetos localizados numa mesma máquina, mas não possuem uma visão global da mesma, o que restringe o acesso somente a seus próprios dados.

O grafo da aplicação representa a distribuição lógica dos seus objetos. Somente através dela não é possível se determinar em quais máquinas estão localizados os objetos. A distribuição física se dará de forma independente do grafo do programa, como mostra o exemplo da FIGURA 3.39, onde os nodos básicos representados na linguagem visual são traduzidos para objetos executando fisicamente em diferentes máquinas.

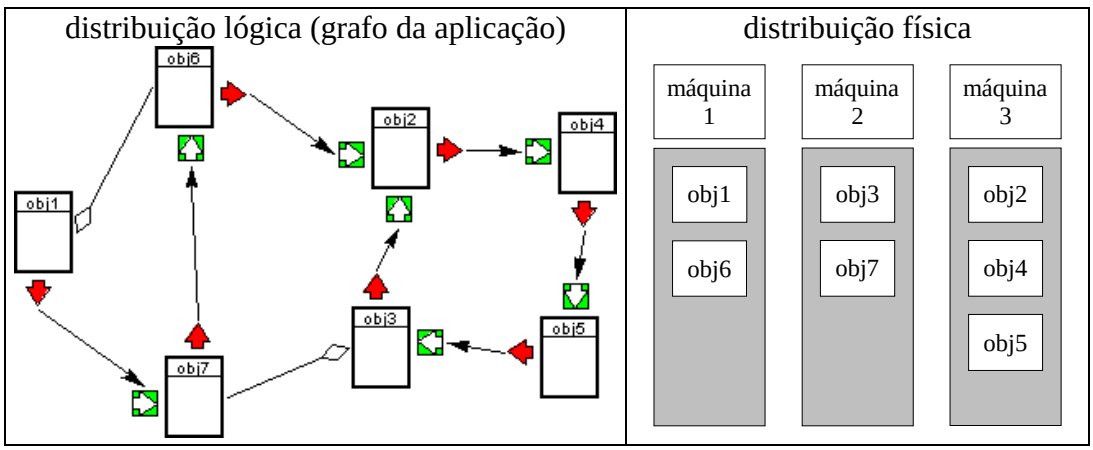


FIGURA 3.39 - Distribuição lógica e física.

Como foi dito, embora os objetos possam estar na mesma máquina, eles estão restritos a áreas de endereçamento individuais. Para dar maior flexibilidade às aplicações, a linguagem oferece a possibilidade de que dois objetos compartilhem memória. Nesse caso, cada um possui uma referência para o outro, de modo que eles

possam realizar chamadas de métodos diretamente e, caso os dados sejam públicos, ter acesso direto a eles. Na implementação, o compartilhamento ocorre somente entre objetos localizados numa mesma máquina. Poderia ser utilizada uma implementação de memória compartilhada distribuída (DSM [FEL94]), mas essa tecnologia não se apropria muito à programação distribuída, pois as redes em geral são maiores e a distância entre os objetos prejudica o desempenho. No exemplo da figura acima, os objetos *obj1* e *obj6* compartilham memória, assim como *obj3* e *obj7*. Desse modo eles podem ter acesso direto aos métodos, fazendo uma invocação local do método.

Outro ponto importante desta linguagem é a instanciação dinâmica de objetos. É uma questão difícil de se expressar visualmente, pois o instante de criação do objeto remoto é, na maioria das vezes, decidido somente em tempo de execução. Enquanto a aplicação está sendo desenvolvida, pouco ou nada pode ser informado a esse respeito. Uma solução é usar uma linguagem especial de declarações, onde o usuário especifica em tempo de projeto uma série de regras que serão testadas continuamente e que num determinado momento podem resultar na criação do objeto remoto. Esse tipo de solução é usado em algumas ferramentas de programação visual paralela, mas tem a desvantagem de restringir a flexibilidade da programação, visto que esta deve seguir o padrão exigido pela linguagem. A solução encontrada pelo trabalho proposto aqui foi a de dar ao programador a responsabilidade da escolha do momento de criação do objeto, permitindo que o usuário insira na aplicação o fragmento de código que irá criar o objeto. Essa solução é muito mais flexível e clara ao programador, pois ele saberá exatamente quando o objeto será criado. A sua desvantagem, entretanto, é a de impossibilitar a visualização do tempo de criação do objeto no editor de grafos, pois esta informação está presente apenas no código fonte textual.

3.4.25.2 Comunicação

A comunicação entre os objetos físicos é expressa visualmente através dos arcos. É dessa maneira que a comunicação é representada pela grande maioria das ferramentas visuais que tratam as aplicações paralelas e distribuídas como grafos. Segundo a linguagem apresentada aqui, a comunicação física entre dois objetos pode se dar através de três maneiras distintas:

- **troca explícita de mensagens**: há muitos algoritmos paralelos e distribuídos baseados nesse paradigma de comunicação. É uma forma de comunicação que se beneficia muito da programação visual, visto que seu tipo de representação gráfica é de certa maneira padronizado (um nodo emissor conectado ao nodo receptor através de uma seta);
- **invocação de método remoto**: ao contrário da troca explícita de mensagens, não há uma representação visual padronizada. Isto leva a que diferentes usuários tenham diferentes interpretações de sua semântica;
- **memória compartilhada**: dois objetos numa mesma máquina podem compartilhar o mesmo espaço de memória.

Vale ressaltar que esses não são os únicos tipos possíveis de comunicação. Como foi falado, é possível se adicionar novos tipos de componentes e assim incrementar as capacidades de comunicação da ferramenta.

A visualização da comunicação possui alguns problemas. A expressão da concorrência é mais simples de se fazer do que a expressão dos relacionamentos, pois a simples distribuição dos elementos no espaço é suficiente para que o usuário perceba a

distribuição. A comunicação também é possível de se representar, mas isso é feito com mais dificuldade. Infelizmente, a representação simples de um arco não é capaz de expressar todos os detalhes do relacionamento. Dois arcos podem ter a mesma representação visual, mas significados ligeiramente diferentes. Esse é o preço que se paga pela simplicidade na visualização, pois um detalhamento excessivo poderia deixar o grafo com muitos elementos e um aspecto visual pesado. Por causa disso, por exemplo, uma invocação remota de método possui uma representação muito semelhante à de uma troca de mensagem.

Uma questão bastante discutida em seções anteriores é a dificuldade de se representar invocação remota de método e criação remota de objeto. Como visto, a representação para esses tipos de relacionamentos exige muitos elementos gráficos. Em cada caso deve haver três arcos e três nodos, sendo que um deles contém ainda mais um componente. A representação ideal talvez fosse a existência de apenas dois nodos e de um arco que indicasse que entre eles haveria a invocação remota. Entretanto, como também foi visto, isso simplificaria a visualização, mas sacrificaria o poder de expressão.

Outro problema das representações visuais é que elas são subjetivas. O significado que uma representação tem para um usuário pode não ser o mesmo para outro. O que esta linguagem oferece para resolver essa questão é a flexibilidade nas representações visuais. Não é definida previamente a representação para nenhum tipo de objeto. O usuário pode assim definir as suas próprias representações visuais, dando-lhes o aspecto que for mais adequado às suas necessidades e ao seu gosto pessoal.

3.4.26 Replicação

Atualmente, esta ferramenta não possui suporte direto à replicação de objetos. Essa situação pode mudar com a integração de outros trabalhos, resultando numa implementação que produziria diferentes impactos na representação visual dos programas:

- **replicação implícita**: o ambiente de execução seria modificado para suportar replicação de objetos de forma transparente para o usuário. Neste caso, a representação visual dos programas permaneceria a mesma, já que foi apenas alterado o ambiente de execução e os objetos não sofreriam alterações. Este trabalho está sendo desenvolvido em [FER2000];
- **replicação explícita**: um novo tipo de nodo básico que implemente por si só a replicação deve ser criado. A replicação então passa a ser visível diretamente no grafo da aplicação.

Em termos de representação visual, a segunda opção é mais expressiva do que a primeira. Contudo, ela deve ser especificada pelo usuário, o que não é necessário no primeiro caso. Uma solução híbrida também pode ser criada onde a replicação é implícita, mas o usuário utiliza objetos com a representação visual diferenciada.

3.5 Ambiente de execução

A execução das aplicações paralelas e distribuídas se caracteriza pela existência de um *middleware* responsável pelas atividades de baixo nível nesses ambientes. O *middleware* cuida da distribuição dos processos e objetos e oferece meios para que eles se comuniquem, independentemente da sua localização. Por exemplo, para a programação paralela, existem os ambientes PVM e MPI que simulam uma máquina

virtual sobre uma rede de processadores, implementando toda a parte de gerenciamento de baixo nível de mensagens e de processos. Isto permite que o usuário tenha uma visão transparente do seu ambiente de execução e a sensação de que todos os seus processos estão localizados na mesma máquina.

Portanto, assim como as ferramentas de programação visual paralela geram código para os ambientes de execução PVM ou MPI para que suas aplicações possam funcionar, esta ferramenta também utiliza um *middleware*. A execução das aplicações assume que o *middleware* já está funcionando em todas as máquinas que forem necessárias para a instanciação dos objetos.

Para este trabalho foi escolhida uma plataforma de distribuição já pronta e amplamente usada chamada Voyager [GLA99]. Inicialmente se pensou no projeto e modelagem de um ambiente próprio de execução distribuída, que deveria ter as principais características necessárias a um ambiente desse tipo, ou seja, invocação remota de métodos e instanciação remota de objetos. Outras funcionalidades também deveriam estar presentes, como por exemplo um serviço de nomes. Entretanto, considerando que o ambiente de execução não é o objetivo principal deste trabalho, tomou-se a decisão de se utilizar um ambiente de objetos distribuídos já pronto. Esta escolha trouxe muitos benefícios, não apenas pela redução na carga de implementação, mas também pela grande quantidade de recursos oferecida por Voyager.

Um ambiente de objetos distribuídos é interessante e poderoso, mas nem sempre está disponível. Isto vale, por exemplo, num ambiente onde a comunicação é feita por meio de *sockets*. Além de permitir que uma aplicação sobreviva na rede sem o uso de um ambiente especial de execução distribuída, um meio de comunicação desse tipo possibilita que uma mesma aplicação utilize diferentes ambientes de execução. Diferentes objetos em diferentes máquinas virtuais distribuídas podem se comunicar através de *sockets*, que é um meio de comunicação "canônico", disponível em praticamente todas as plataformas.

O ambiente de execução utilizado não é definitivo. Para usar outra plataforma de execução, basta que o usuário implemente um novo tipo de nodo básico e junto com ele o gerador de código que suporte o novo ambiente. Outro ambiente suportado pela implementação da ferramenta é o RMI.

3.6 Resumo do capítulo

Este capítulo apresentou o modelo da ferramenta, onde inicialmente se discutiram os tipos de usuários e de aplicações a que a ferramenta se destina para que em cima disso fosse feita uma análise que chegasse aos objetivos e requisitos que ela deveria possuir. Com base nisso, a interface gráfica da ferramenta foi estruturada em três partes principais: a janela principal que contém a paleta de componentes, o editor de propriedades e eventos e a janela do grafo onde é feita a edição das partes visual e textual das aplicações.

As aplicações são estruturadas em grafos. O usuário os cria e os combina através do editor de grafos, utilizando para isso as ferramentas e objetos disponíveis na paleta de componentes. Ao mesmo tempo em que edita visualmente a estrutura da sua aplicação, o usuário insere o código em Java com a lógica da sua aplicação.

3.6.1 Resumo da linguagem visual

Como uma forma de dar uma visão geral da linguagem visual utilizada nesta ferramenta, serão apresentadas duas tabelas contendo os tipos de nodos e arcos existentes, com uma definição rápida e alguns exemplos. A TABELA 3.12 mostra os nodos e a TABELA 3.13, os arcos.

TABELA 3.12 - Nodos.

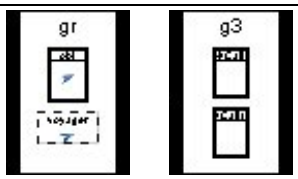
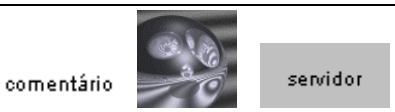
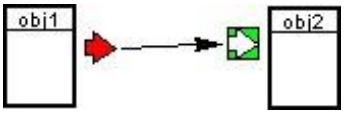
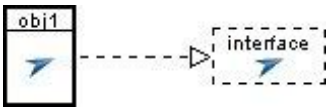
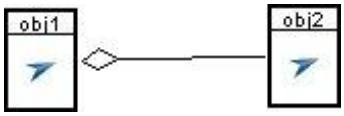
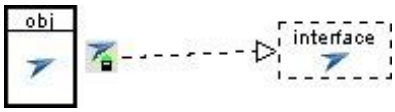
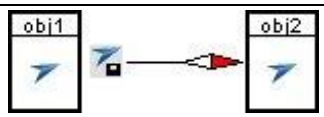
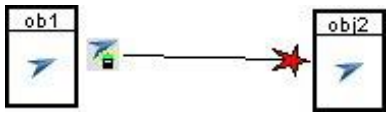
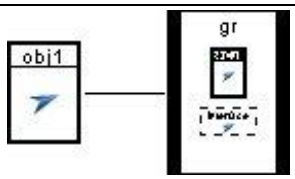
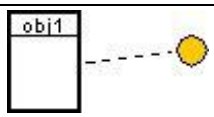
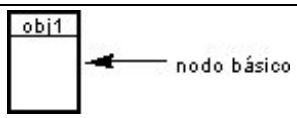
objeto	definição	exemplo(s)
nodo básico	objeto com código editável	
nodo de interface	interface de objeto remoto	
nodo terminal	classe ou interface já prontas em forma de código objeto	
nodo virtual	recurso externo	
componente	componente reutilizável	
procurador	referência local a objeto remoto	
criação dinâmica	instanciação de objeto remoto	
método	método de classe ou interface	
interface de grafo	parte visível de grafo	
chamada a grafo	representação de um grafo completo	
documentação	documentação e apresentação	

TABELA 3.13 - Arcos.

arco	descrição	exemplo(s)
normal	mapeamento de propriedades	
implementação de interface	nodo básico implementa interface	
memória compartilhada	compartilhamento de dados entre dois nodos básicos	
interface de objeto remoto	especificação de interface para componentes procuradores ou de criação dinâmica	
referência a objeto remoto	referência local a objeto remoto	
criação de objeto remoto	objeto local cria objeto remoto	
criação de grafo	objeto local cria grafo	
interface de grafo	objeto conectado à interface de grafo é visível externamente	
documentação	comentário visual	

4 Implementação

Este capítulo descreverá os principais aspectos da implementação do modelo proposto no capítulo anterior. Em primeiro lugar serão citadas algumas características gerais da construção de todo o ambiente de programação e, na seqüência, detalhes mais técnicos sobre seus principais módulos e funcionalidades. Ao final do capítulo, a fim de avaliar o uso da ferramenta, serão mostrados exemplos de aplicações desenvolvidas com a mesma.

4.1 Estrutura geral

A estrutura geral da ferramenta está na FIGURA 4.1. Existe um módulo que apresenta os comandos utilizados pelo usuário e a paleta de componentes onde estão os objetos que podem ser inseridos nas aplicações. Existe uma instância de uma janela de edição de propriedades e eventos e diversas instâncias de um editor de grafos, o qual compreende um editor gráfico interativo, uma janela de visualização da hierarquia do grafo e diversos editores textuais do código fonte dos nodos básicos e de interface.

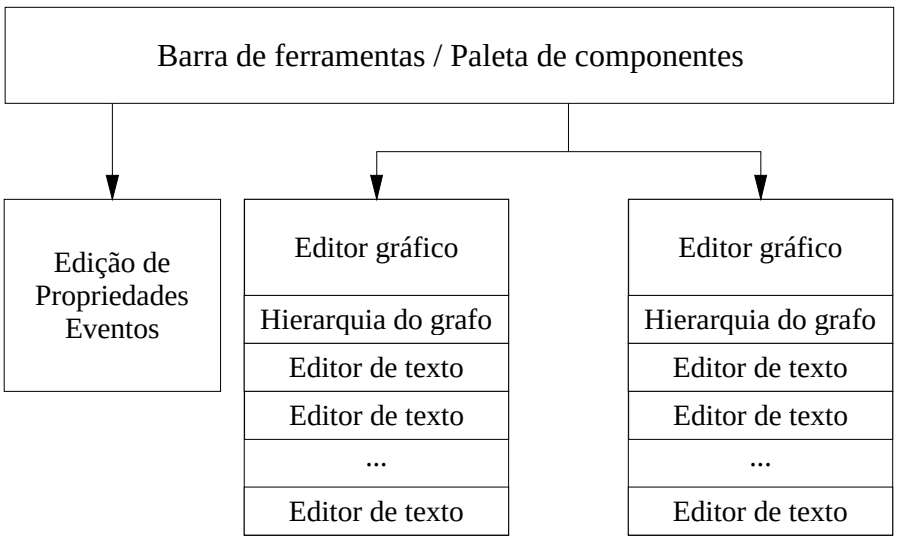


FIGURA 4.1 - Estrutura geral da ferramenta.

Essa estrutura permite que sejam construídos vários grafos simultaneamente, o que é muito útil em aplicações de grande porte, que em geral são subdivididas em diversos grafos. Quanto aos editores de texto, em cada grafo existe um para cada nodo básico ou nodo de interface que dele fizer parte.

A implementação da ferramenta foi feita inteiramente em Java, utilizando-se o Sun JDK 1.2.2 como apoio ao desenvolvimento. A interface gráfica foi construída com as bibliotecas do pacote Swing [SUN2000e], disponível no próprio JDK utilizado. Também neste JDK estão todas as classes necessárias para a manipulação dos componentes JavaBeans, muito importantes nesta implementação. Para a edição do código foram utilizados apenas um editor de textos e um compilador de linha de comando. No início dos trabalhos foi estudado o uso de ferramentas de implementação mais poderosas, como o JBuilder. Entretanto, verificou-se que essas ferramentas tornavam o código fonte da aplicação muito grande e adicionavam pacotes proprietários, que deveriam ser incluídos numa futura distribuição. Outros fatores como

o fato do JBuilder ser pago e de necessitar de *hardware* poderoso para executar com eficiência também pesaram nessa decisão.

Para economizar tempo de desenvolvimento, algumas bibliotecas prontas foram incorporadas. Parte do gerenciamento dos componentes JavaBeans foi estudada a partir do BDK (*Beans Development Kit*) [SUN2000d] e o realce da sintaxe Java foi conseguido através da inclusão de código fonte desenvolvido originalmente para a ferramenta JIPE [LAW2000].

4.1.1 Linguagem de programação Java

O ambiente de programação escolhido tanto para a construção da ferramenta como para a interface com o usuário foi Java. Java é um ambiente de computação bastante abrangente, com muitas características importantes para a realização deste trabalho:

- linguagem orientada a objeto, possuindo uma rica biblioteca de classes que facilita tanto a implementação da ferramenta como das aplicações nela construídas;
- bibliotecas para geração de interfaces gráficas: o pacote Swing contém as classes mais importantes para este fim (botões, menus, janelas, caixas de texto etc);
- suporte especial à programação orientada a eventos;
- arquitetura padronizada de componentes de *software* (JavaBeans);
- portabilidade de código objeto;
- suporte à comunicação e segurança em ambiente de rede;
- tecnologia de objetos distribuídos: RMI e Voyager (esta última desenvolvida por terceiros);
- existência de ferramentas de geração automática de documentação de código, como por exemplo o javadoc [SUN2000c].

A deficiência principal de Java, o baixo desempenho, não é fator muito importante neste trabalho e por isso não é suficiente para evitar a escolha de Java. Caso o desempenho fosse um item crítico, certamente outros ambientes e linguagens de programação deveriam ser melhor levados em conta.

4.1.2 Ambiente de execução

Quando são usados objetos distribuídos, a execução de uma aplicação gerada com essa ferramenta pressupõe a execução prévia de um ambiente de controle. Java, por si só, não apresenta a possibilidade de fazer chamadas remotas a objetos e, por isso, foram escolhidos para suportar a execução dos objetos dois ambientes: Voyager e RMI. O primeiro é desenvolvido por uma empresa privada e possui a maioria das funcionalidades necessárias a objetos distribuídos, enquanto RMI é um sistema mais simples e está disponível nas versões mais recentes da biblioteca Java, razões que motivaram a escolha de seu uso nesta ferramenta.

Quanto à escolha de Voyager, outras possibilidades foram analisadas, como os ambientes Mitsubishi Concordia e IBM Aglets. Entretanto, as principais características de Voyager decidiram a escolha a seu favor:

- simplicidade de uso;
- portabilidade: 100% Java;
- custo zero: o uso tanto comercial como pessoal da versão mais simples (mas ainda adequada a este trabalho) é livre de taxas;
- já é utilizado nos meios acadêmicos. Outros projetos em desenvolvimento no GPPD-II UFRGS estão sendo desenvolvidos em Voyager;
- leveza: o custo adicional de desempenho imposto à aplicação é pequeno.

Outra razão da escolha de Voyager em adição a RMI é o fato dele instanciar remotamente objetos. Na versão utilizada, RMI oferece invocação remota de objetos, mas ainda não possibilita instanciação remota de objetos, exigindo que os objetos sejam disparados manualmente nas máquinas remotas ou outro sistema como Voyager seja usado.

Informações mais detalhadas sobre Voyager já foram apresentadas na seção 2.5.4, a qual tratou das suas características existentes para objetos distribuídos.

4.1.3 Estrutura de pacotes

Assim como o JDK, a maioria do *software* desenvolvido em Java é organizada em pacotes. Cada pacote contém classes que se assemelham na sua finalidade, facilitando a sua busca e utilização. A necessidade de se estruturar as classes em pacotes é diretamente proporcional à sua quantidade e, em casos como o deste trabalho, que contém mais de 250 classes, dividi-las em pacotes é quase uma obrigatoriedade.

br.ufrgs.inf.gppd.dobuilder	base do pacote
ide	interface gráfica da ferramenta
editors	editores de propriedades
gui	interface gráfica
dialog	janelas de diálogo da ferramenta
tools	classes auxiliares
vpj	objetos da linguagem visual
base	nodos básicos
service	nodos básicos de serviço
codegen	geradores de código
component	componentes
gui	componentes de interface gráfica
port	portas de comunicação
documentation	nodos de documentação
dynamic	componentes de criação dinâmica
event	classes de gerenciamento dos eventos
graph	grafos
dialog	janelas de configuração dos grafos
graphcall	nodos de chamada a grafo
graphinterface	nodos de interface de grafo
interfaces	nodos de interface
method	métodos
proxy	componentes procuradores
terminal	nodos terminais
virtual	nodos virtuais

FIGURA 4.2 - Estrutura de pacotes da implementação da ferramenta.

Assim, as classes desenvolvidas para este trabalho foram organizadas na estrutura de pacotes da FIGURA 4.2. O prefixo inicial dos pacotes (br.ufrgs.inf.gppd) vem do costume que se tem de nomeá-los de acordo com o local onde foram desenvolvidos.

A classe com o código *main* da ferramenta é a ide.gui.IDE. Assim, para executar a ferramenta, deve-se passar como parâmetro para o interpretador Java a classe ide.gui.IDE.

4.2 Editor de grafos

O editor de grafos não é igual aos editores de grafos tradicionais, onde se tem apenas uma representação visual de um objeto. Neste editor, os objetos inseridos são realmente instanciados no grafo. Quando se insere um nodo básico, por exemplo, não está se inserindo apenas uma representação do objeto, mas uma instância da classe do objeto. De outra forma não seria possível uma análise para buscar as propriedades e permitir que cada objeto implementasse a forma como seria visualizado no grafo.

Esse tipo de implementação não é nenhuma novidade e é comum aos editores gráficos das ferramentas que manipulam JavaBeans. Assim, antes de se iniciar a sua implementação, foi estudado o pacote BDK, que contém os códigos fontes de uma ferramenta que também lida com essa arquitetura de componentes. Boa parte deste editor de grafos, principalmente a parte que trata os JavaBeans, foi baseada no código disponibilizado com o BDK.

4.2.1 Estruturas de dados

Internamente ao editor, são usadas duas estruturas de dados principais na sua implementação:

- **árvore**: armazenamento dos nodos do grafo;
- **vetor**: lista dos arcos.

A árvore é utilizada para representar a relação de composição existente entre os objetos do grafo. É importante se notar que isso não significa que não possa haver ciclos no grafo da aplicação, isto é, que não possa haver arcos que saem de um objeto e chegam nele mesmo. O que não pode acontecer é um ciclo no grafo de composição. Assim não é possível, por exemplo, que um nodo básico contenha um componente que por sua vez contenha esse mesmo nodo básico. O componente pode conter um nodo básico da mesma classe, mas a instância não pode ser a mesma do objeto que o contém.

Os objetos armazenados nessa árvore têm todos os dados necessários para que o objeto do usuário seja reconstruído dentro do grafo. As principais delas são os valores das propriedades, os métodos tratadores dos eventos e a posição do objeto no grafo.

O vetor é a forma utilizada para se representar os arcos. Há uma lista de arcos que armazenam dentro de si a informação de qual nodo é a fonte e qual é o destino do arco, os quais são referências aos objetos da árvore citada acima.

A árvore e o vetor são as estruturas de dados salvas em disco quando se deseja abrir o mesmo grafo em outra sessão. Com as facilidades de serialização de objetos em Java [SUN2000e], essa tarefa fica bastante simples, pois não é necessário se criar um formato especial de armazenamento. Basta apenas serializar as estruturas de dados e gravá-las em arquivo, possibilitando que posteriormente sejam restauradas diretamente para os seus conteúdos originais.

4.2.2 Organização

A organização do editor de grafos é baseada numa máquina de estados. À medida que o usuário seleciona os componentes e realiza movimentos e cliques com o mouse, a máquina vai mudando de estado e as ações correspondentes vão sendo tomadas. Para cada estado são definidas as entradas, ações e transições possíveis. Por exemplo, quando um objeto está selecionado, é permitido ao usuário entre outras coisas mover o mouse e pressionar a tecla *delete*, que correspondem às ações de reposicionamento e remoção do objeto, respectivamente.

4.2.3 Hierarquia do grafo

A hierarquia do grafo é uma representação gráfica da árvore em que a aplicação é estruturada. A biblioteca de Java oferece a classe JTree [SUN2000e] que implementa a apresentação de uma árvore construída segundo um determinado formato específico, também existente na biblioteca de Java. Como a árvore utilizada seguiu essa forma, a sua apresentação é bem simples de se fazer com um objeto dessa classe.

4.2.4 Editor textual

Java oferece em sua biblioteca algumas classes para edição de texto. Essas classes, entretanto, possuem apenas as funcionalidades mais básicas de edição. Para acrescentar alguns recursos importantes, na implementação do editor de texto foi utilizada uma classe já pronta, buscada na Internet [LAW2000], que além das principais funções de edição de texto realiza também o realce de sintaxe em Java (diferenciação através de cores entre os diferentes objetos léxicos do programa, como palavras reservadas e constantes). Com essa característica, a apresentação do código fonte fica consideravelmente melhor, como se percebe na FIGURA 4.3.

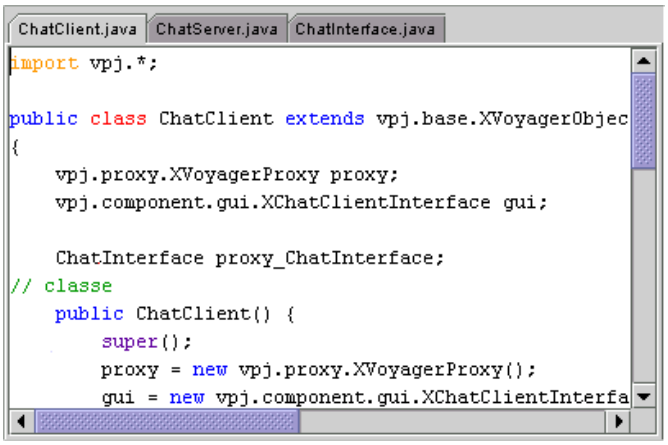


FIGURA 4.3 - Editor de texto com realce de sintaxe em Java.

Muitos melhoramentos poderiam ser feitos em cima dessa versão inicial da edição do texto, como por exemplo a apresentação do número da linha que está sendo editada. Entretanto, por questão de definição de prioridades na implementação, estes avanços foram deixados para versões futuras da ferramenta.

4.3 Barra de ferramentas e janela de propriedades

A FIGURA 4.4 apresenta a parte da interface gráfica da ferramenta onde estão a barra de ferramentas com botões de comandos gerais (compilar, salvar arquivo etc), os objetos da linguagem visual e a saída dos comandos de compilação e execução.

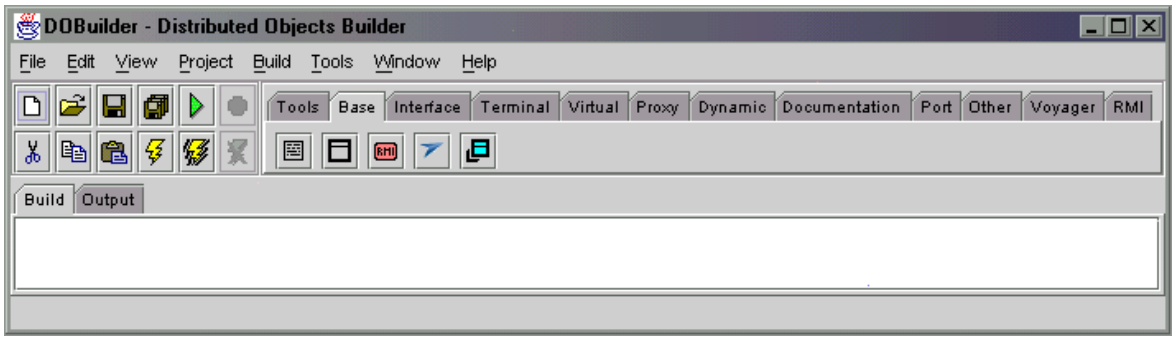


FIGURA 4.4 - Barra de ferramentas e saída de comandos.

Outra peça muito importante da ferramenta é a janela de propriedades e eventos, onde são editados e visualizados os métodos tratadores de eventos e as propriedades dos objetos. Para que essas informações sejam editadas pelo usuário, elas são manipuladas pela ferramenta através da biblioteca Java que oferece um pacote inteiro (java.beans) para a manipulação de componentes. Com as classes desse pacote, uma ferramenta pode inspecionar o código compilado das classes e obter suas propriedades e eventos, oferecendo então editores para armazenar e recuperar tais informações, como mostra a FIGURA 4.5.

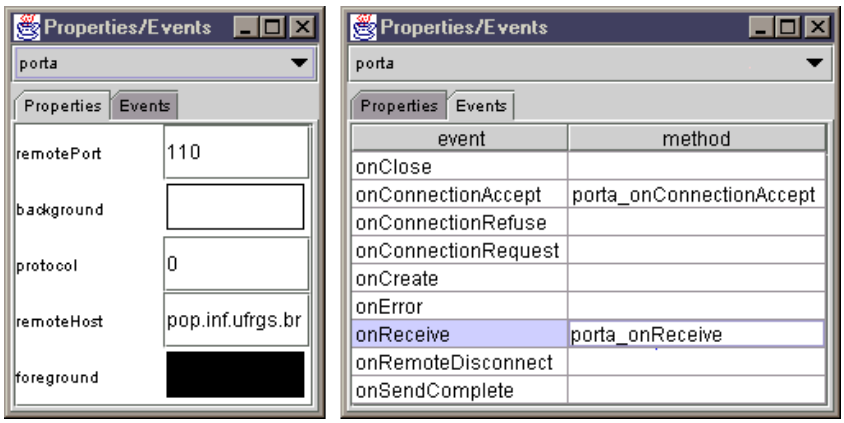


FIGURA 4.5 - Propriedades e eventos do componente porta de chamada e serviço.

Os editores são definidos pelo tipo de dado editado. Por exemplo, quando os valores são numéricos ou textuais, surge uma caixa de texto comum e, quando a propriedade é uma cor, surge uma caixa de diálogo com os controles característicos para a escolha de uma cor.

Quando da inserção de um novo componente ou da seleção de um já existente, a ferramenta faz uma análise do objeto e a apresenta na janela de propriedades e eventos. As alterações que o usuário fizer nesta janela são refletidas imediatamente no objeto e no código fonte, se for o caso.

4.4 Componentes de *software*

Todos os objetos da linguagem visual são implementados como componentes JavaBeans. Esta padronização facilita principalmente a integração de novos componentes, permitindo que características extras sejam adicionadas ao sistema sem a necessidade de alteração na sua estrutura. Se uma determinada funcionalidade não existe, em muitos casos, apenas com a implementação de um novo componente é possível se resolver o problema.

As vantagens da utilização do modelo de componentes de *software* já foram expostas na seção 2.4. Para uma melhor compreensão da implementação deste modelo nesta ferramenta, será apresentado a seguir um exemplo de componente.

4.4.1 Exemplo de componente

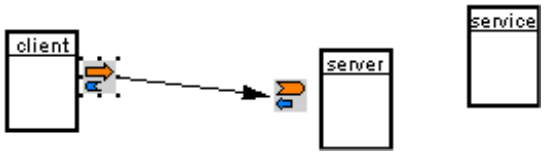


FIGURA 4.6 - Porta de chamada de serviço.

O componente apresentado como exemplo é o componente porta de chamada de serviço (implementa a conexão a serviços remotos TCP – está selecionado na FIGURA 4.6). Assim como todos os outros componentes, este é definido segundo propriedades, métodos e eventos. As suas características são apresentadas na TABELA 4.1.

TABELA 4.1 - Propriedades, métodos e eventos do componente.

propriedades	**métodos**	**eventos**
`remotePort`	`open`	`onCreate`
`remoteHost`	`close`	`onClose`
`protocol`	`receive`	`onConnectionRequest`
`foreground`	`receiveAsync`	`onConnectionAccept`
`background`	`send`	`onConnectionRefuse`
	`sendSync`	`onReceive`
		`onSendComplete`
		`onError`
		`onRemoteDisconnect`

As propriedades são características que podem ser editadas em tempo de projeto através dos editores oferecidos pela ferramenta ou em tempo de execução através de chamadas inseridas no código fonte. Cada propriedade possui um método que lhe atribui um valor e outro que o recupera. Por exemplo, para a propriedade `remotePort`, o método `getRemotePort` retorna o valor da propriedade e o método `setRemotePort` lhe dá um novo valor (este é um dos *design patterns* dos componentes JavaBeans).

Os métodos são tratados da mesma maneira que métodos tradicionais de qualquer objeto Java, já que os componentes também são objetos Java. Os eventos mostrados na TABELA 4.1 são os eventos gerados por este componente. Um outro objeto que estiver interessado nesses eventos deve se registrar neste objeto para que ele o notifique quando o evento for gerado (a implementação do modelo de eventos de Java está explicada com detalhes na seção 2.3.2).

Como está descrito na seção 4.3, as propriedades e os nomes dos métodos podem ser editados na janela de propriedade e eventos, que apresenta editores específicos para estes fins. As alterações dos valores de propriedades nesses editores, quando for o caso, refletem-se imediatamente na apresentação visual do componente e no código fonte do nodo básico correspondente. A FIGURA 4.7 mostra como as características do componente são tratadas no código fonte.

```
import vpj.*;
import vpj.event.*;

public class NodoBasico extends vpj.base.XDistributedObject {
    vpj.component.port.XCallPort porta;

    public NodoBasico() {
        super();
        porta = new vpj.component.port.XCallPort();
        porta.setRemotePort(110);
        porta.setRemoteHost("pop.inf.ufrgs.br");
        porta.addXCallPortListener(new XCallPortAdapter() {
            public void onConnectionAccept(XCallPortEvent e) {
                porta_onConnectionAccept(e);
            }
            public void onReceive(XCallPortEvent e) {
                porta_onReceive(e);
            }
        });
    }

    public void porta_onConnectionAccept(XCallPortEvent e) {
        System.out.println("Conectado.");
        porta.send("mensagem");
    }

    public void porta_onReceive(XCallPortEvent e) {
        String msg = (String) porta.receive();
        System.out.println("Mensagem recebida = " + msg);
    }

    public void run() {
        System.out.println("Conectando ao servidor ...");
        porta.open();
    }
}
```

FIGURA 4.7 - Código do nodo básico que contém a porta de chamada de serviço.

A parte cinza mais escura representa o código que altera as propriedades do componente `porta`, que é da classe `XCallPort`. A parte cinza clara corresponde aos eventos gerados pelo objeto que devem ser tratados pelo nodo básico. Os eventos nesse caso são `onConnectionAccept` e `onReceive`.

Quanto à representação visual, não é obrigatório que um componente possua uma representação visual específica. Entretanto, como esta ferramenta é de programação visual, todos os componentes liberados com ela possuem essa característica. Novos componentes que não tenham uma representação visual também podem ser adicionados à ferramenta, mas aconselha-se que, sempre quando for possível, os componentes ofereçam essa possibilidade.

4.5 Funcionamento do editor de grafos

Conforme o tipo de objeto a ser inserido ou relacionamento a ser criado, diferentes ações são tomadas. Algumas atividades são comuns à inserção de todos os objetos: atualização da estrutura de dados contendo os objetos, inserção do objeto no editor gráfico e atualização da janela de hierarquia de objetos. Outras, entretanto, são específicas a seu caso, como mostram a TABELA 4.2 e a TABELA 4.3, que descrevem as ações para cada comando do usuário na inserção de objetos e de relacionamentos, respectivamente.

TABELA 4.2 - Inserção de objetos.

objeto	ações
nodo básico	criação de uma nova janela para edição do código, com criação do código inicial da classe ou da interface
nodo de interface	
nodo terminal	se já foi definida uma classe base para este nodo terminal, carrega o objeto e o apresenta no editor já com todos os seus componentes
chamada a grafo	se já está definido o nome do arquivo que contém o grafo, carrega esse grafo e apresenta os objetos que fazem parte da interface do grafo dentro do nodo de chamada a grafo
componente	atualização do código fonte do nodo básico pai para conter a declaração do componente
procurador	
criação dinâmica	
método	atualização do código fonte do nodo básico ou nodo de interface que o contém
nodo virtual	nenhuma ação especial, além da inserção do objeto no grafo, como em todos os outros casos
interface de grafo	
documentação	

TABELA 4.3 - Criação de relacionamentos.

arco	ações
normal	apresenta a janela de mapeamento de propriedades para que o usuário defina os relacionamentos. Se o objeto fonte já possuir mapeamentos predefinidos para o objeto destino, os mapeamentos são feitos automaticamente. Os códigos fontes dos nodos básicos que contêm os componentes são atualizados quando as propriedades são alteradas
referência a objeto remoto	
criação de objeto remoto	
implementação de interface	atualização do código fonte do objeto com a declaração da implementação da interface e com a implementação dos métodos declarados na interface
memória compartilhada	alteração do código fonte do objeto pai para conter a declaração e criação do objeto filho
interface de objeto remoto	adição ao nodo básico pai de métodos especiais: para a referência ao objeto remoto, um método que retorna essa referência; para a criação dinâmica, um que faz a criação remota; para a criação de grafo, um que cria o grafo
criação de grafo	
interface de grafo	nenhuma ação especial, além da inserção do arco no grafo, como acontece nos outros casos
documentação	

A implementação dos relacionamentos leva em conta a TABELA 3.8, que mostra que tipo de relacionamento pode ocorrer entre dois objetos quaisquer. Quando o usuário tenta inserir um arco, o editor verifica que tipos de objetos eles são e seleciona o tipo de arco que será inserido. Se o relacionamento não for possível, nenhum arco será criado.

4.6 Implementação dos objetos da linguagem visual

Como foi visto na descrição do modelo da linguagem visual no capítulo 3, há vários tipos de objetos que podem ser introduzidos na aplicação. A partir de agora será apresentada a organização da implementação desses componentes.

4.6.1 Estrutura

Cada tipo de objeto da linguagem é representado na implementação por uma interface Java, como está na FIGURA 4.8. Embora não seja uma notação formal, a representação da figura citada cumpre o seu objetivo de exibir as relações existentes entre as interfaces da linguagem.

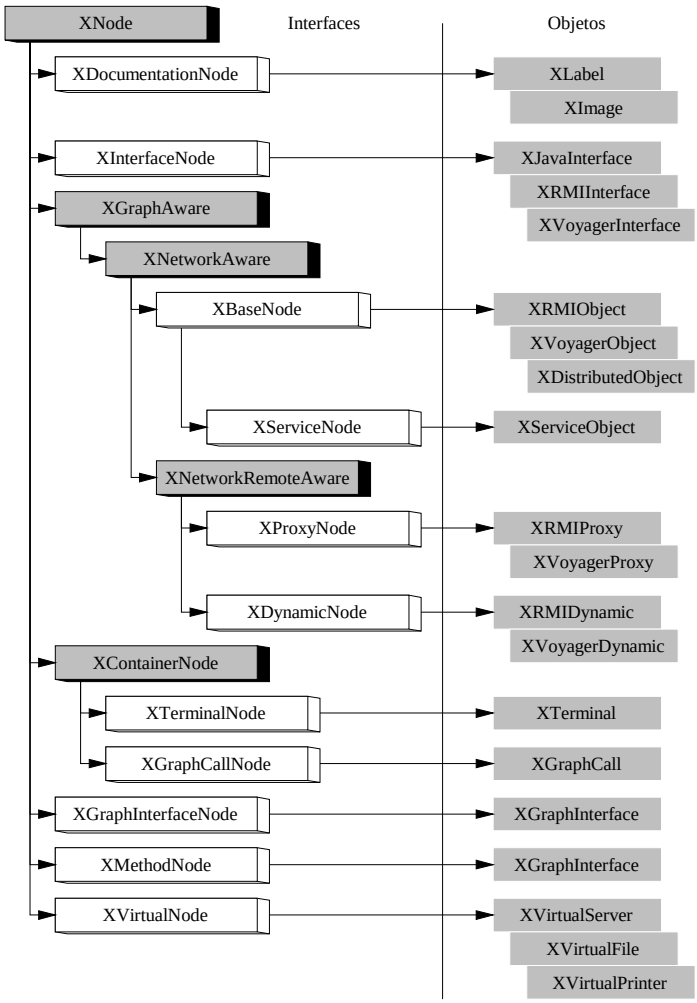


FIGURA 4.8 - Estrutura de objetos da linguagem visual.

Os retângulos à esquerda representam interfaces Java, enquanto os retângulos da direita indicam alguns dos objetos que implementam essas interfaces. Por exemplo, `XBaseNode` é a interface implementada pelo nodo básico `XVoyagerObject` (como será visto posteriormente, deve ser implementada por qualquer nodo básico). Pelo diagrama, vê-se que essa interface deriva de `XNetworkAware`, de `XGraphAware` e da de mais alto nível, `XNode`. Sendo assim, ao implementar a interface `XBaseNode`, o nodo básico deverá também conter os métodos de todas essas outras interfaces, recebendo de cada uma delas o comportamento correspondente.

O motivo de se utilizar somente interfaces e não classes para se estruturar os objetos da aplicação se deve ao fato de que, ao contrário de outras linguagens de programação orientada a objeto, Java não permite a herança múltipla de classes. Se `XBaseNode`, por exemplo, fosse uma classe e não uma interface, caso o usuário quisesse criar um outro nodo básico, a única possibilidade existente seria a de ele derivar a sua classe a partir desta, não podendo fazer o mesmo a partir de outra classe. Sendo `XBaseNode` uma interface, ele faz simplesmente com que sua classe implemente os métodos dessa interface sem se preocupar com as limitações da herança de Java.

Entretanto, para manter a mesmas facilidades das superclasses, que já implementam alguns métodos, foram implementadas algumas classes auxiliares, como `XDefaultBaseNode`. Se o usuário não precisa derivar de outra classe, basta que ele derive seu nodo básico de `XDefaultBaseNode` e implemente somente os métodos que deseja alterar.

4.6.2 Interfaces básicas

As interfaces destacadas no diagrama da FIGURA 4.8 (`XNode`, `XGraphAware`, `XContainerNode`, `XNetworkAware`, `XNetworkRemoteAware`) são interfaces que servem de base para a construção de outras interfaces. Embora fosse tecnicamente possível, os componentes desenvolvidos nessa ferramenta não implementam de forma direta nenhuma delas. Elas funcionariam como se fossem classes abstratas, mas que não foram implementadas dessa maneira pelos motivos já delineados há pouco na seção anterior.

É importante ressaltar que nem todos os métodos pertencentes a uma interface são relevantes para um objeto que a implemente. Embora o método exista na interface, ele pode não ser efetivamente usado por um objeto. A razão de ele existir na interface é a generalização. A fim de evitar interfaces muito específicas, dedicadas somente a um ou outro tipo de objeto, foram criadas interfaces mais abrangentes.

A seguir serão apresentados com mais detalhes os métodos e a finalidade das interfaces chamadas básicas nesse contexto.

4.6.2.1 XNode

A interface de qual todas as outras derivam é a interface `XNode`. Ela apresenta métodos comuns a todos os objetos e serve para identificar um objeto como sendo um componente desenvolvido para esta ferramenta.

TABELA 4.4 - Métodos da interface XNode

método	descrição
`[get|set]ParentNode`	obtém/define o objeto pai
`paintArc[To|From]`	métodos para desenhar os arcos de entrada e de saída, quando o objeto realizar conexões

Como mostra a TABELA 4.4, dois conjuntos de métodos existem nessa interface. O primeiro deles é usado no gerenciamento do objeto pai de qualquer objeto. A referência ao objeto pai deve existir porque em certos casos é necessário se obter informações que só estão nele, como no caso de um componente procurador que busca o identificador da máquina em que se encontra.

O segundo conjunto de métodos é usado na visualização dos relacionamentos. Genericamente, quando uma conexão parte de um objeto A em direção a B, o desenho do arco é feito por A, através do método `paintArcTo`. Caso esse método não tenha sido definido em A, é chamado o método `paintArcFrom` no objeto B. Se nem esse existir, a ferramenta provê uma representação padrão para o relacionamento. Em qualquer caso, ao método chamado são passados como parâmetros os tipos e as referências aos objetos que fazem parte do relacionamento. Isso permite que se dê uma representação diferente para cada situação, com base nos objetos participantes do relacionamento.

4.6.2.2 XGraphAware

Os objetos que reconhecem a existência de diferentes níveis de grafos devem implementar a interface `XGraphAware`. O nível de grafo se define como a posição do objeto em relação às chamadas de grafo. Se um objeto não foi criado a partir de uma chamada de grafo, ele está no nível mais alto (nível zero). Se ele foi criado a partir de uma chamada a grafo que estava no nível zero, então ele estará no nível 1 e assim por diante. Os métodos da TABELA 4.5 são usados para atualizar e conhecer o nível do grafo em que um objeto se encontra.

TABELA 4.5 - Métodos da interface XGraphAware

método	descrição
`[get|set]GraphPath`	obtém/define o nível do grafo do objeto

Os objetos que utilizam os recursos providos por essa interface são os nodos básicos, os componentes procuradores e os componentes de criação dinâmica. Como o comportamento desses objetos é influenciado pelo nível de grafo em que se encontram (maiores detalhes na seção 4.8), tais métodos são necessários no momento de se obter uma referência remota ou de se criar um objeto remoto.

No componente procurador, por exemplo, o nível do grafo é utilizado da seguinte maneira: quando um grafo é criado, o método `setGraphPath` é chamado em todos os objetos que implementam a interface `XGraphAware`, os quais passam a ter conhecimento do nível de instanciação do grafo em que se encontram. No caso do componente procurador, essa informação é utilizada no momento da busca à referência ao objeto remoto. Ele irá buscar o objeto remoto que tenha o nome especificado pelo caminho do grafo seguido pelo valor da propriedade que identifica o objeto remoto (`remoteName`). Isto é feito para se diferenciar o mesmo objeto remoto em diferentes níveis do grafo. Como em cada nível do grafo eles recebem o nome juntamente com o caminho do grafo, é possível se buscar exatamente o objeto que se deseja.

4.6.2.3 XContainerNode

Determinados objetos no grafo podem representar um conjunto de objetos, fazendo com que sua apresentação visual deva refletir essa condição. É o caso dos nodos terminais e dos nodos de chamada a grafo.

A forma de implementação desses objetos exige que eles notifiquem o editor de grafos quando alguma alteração for feita em seus estados. Os dois possuem referências para o objeto que eles representam, que no caso do nodo de chamada a grafo é um nome de arquivo e no do nodo terminal é o nome da classe do nodo básico representado por este nodo terminal. Quando essas propriedades forem alteradas, o editor deve atualizar o grafo para refletir a condição real dos objetos.

TABELA 4.6 - Métodos da interface XContainerNode

método	descrição
`[add|remove] PropertyChangeListener`	gerencia a lista de objetos que está interessada nos valores das propriedades

A forma de os objetos notificarem o editor é oferecer métodos para isso. A TABELA 4.6 mostra os métodos pelos quais o editor se registra no objeto, fazendo com que quaisquer alterações nas propriedades dos objetos sejam a ele notificadas.

4.6.2.4 XNetworkAware

Os objetos que reconhecem a sua existência num ambiente distribuído implementam a interface `XNetworkAware`, que oferece os métodos necessários para que o objeto faça parte do contexto da rede (TABELA 4.7).

TABELA 4.7 - Métodos da interface XNetworkAware

método	descrição
`[get|set]Host`	obtém/define o endereço da máquina onde o objeto está ou deva ser instanciado
`[get|set]Name`	obtém/define o nome segundo o qual o objeto é conhecido na rede (usado no servidor de nomes)
`[get|set] NameServerAddress`	obtém/define o endereço do servidor de nomes

Esses métodos são utilizados principalmente pelos nodos básicos. Eles devem ter uma máquina onde são instanciados e uma identificação no servidor de nomes. Essas características derivam da implementação dos *middlewares* de objetos distribuídos utilizados neste trabalho, como Voyager e RMI, que utilizam um servidor de nomes e obtêm as referências remotas através dele.

4.6.2.5 XNetworkRemoteAware

Determinados objetos necessitam realizar operações de rede que exigem o conhecimento do estado de um objeto remoto. É para esses objetos que existe a interface `XNetworkRemoteAware`.

TABELA 4.8 - Métodos da interface XNetworkRemoteAware

método	descrição
`[get|set]RemoteHost`	obtém/define a máquina remota referenciada
`[get|set]RemoteName`	obtém/define o nome utilizado pelo objeto remoto no servidor de nomes

Os objetos que utilizam os métodos da TABELA 4.8 são o componente procurador e o componente de criação dinâmica. Este último utiliza o primeiro método para saber em qual máquina instanciar o objeto e o componente procurador usa o segundo método para buscar a referência ao objeto remoto.

4.6.3 Interfaces utilizadas para criar objetos

Até agora foram vistas as interfaces chamadas básicas, ou seja, aquelas que servem de base para a criação de outras interfaces. Estas outras interfaces são utilizadas diretamente na criação dos objetos da linguagem e serão vistas a partir de agora.

Como já foi visto, cada objeto da linguagem visual é implementado como um componente JavaBean. A ferramenta descobre o tipo do componente analisando as interfaces que ele implementa por meio da introspecção Java sobre os objetos [SUN2000e]. Cada tipo de objeto deve implementar uma interface específica, dando à ferramenta a possibilidade de classificar cada componente e assim controlar as ações sobre eles. Um nodo de interface, por exemplo, deve implementar a interface `XInterfaceNode` para ser encarado pela ferramenta como um objeto desse tipo.

TABELA 4.9 - Interfaces que devem ser implementadas pelos objetos.

Nodo	Interface
nodo básico	`XBaseNode`
nodo de interface	`XInterfaceNode`
nodo terminal	`XTerminalNode`
nodo virtual	`XVirtualNode`
nodo procurador	`XProxyNode`
nodo de criação dinâmica	`XDynamicNode`
método	`XMethodNode`
nodo de interface de grafo	`XGraphInterfaceNode`
nodo de chamada a grafo	`XGraphCallNode`
nodo de documentação	`XDocumentationNode`
componente	-

A TABELA 4.9 mostra a relação entre o objeto da linguagem e a interface que ele deve implementar. Cada interface apresenta métodos que são usados tanto em tempo de projeto como de execução para realizar as funções do tipo de objeto correspondente.

A única exceção em termos de implementação de interface é o objeto do tipo componente, que não implementa nenhuma interface em especial. Isso permite que componentes desenvolvidos em outras ferramentas possam ser integrados a esta ferramenta de forma simples, sem restrições e inclusive sem alterações no código. Portanto, qualquer objeto que não seja classificado pelas interfaces que implementa será um objeto do tipo componente.

A partir de agora serão descritas as interfaces utilizadas para construir os componentes. A apresentação dos métodos dessas interfaces contribuirá muito para a compreensão do significado e do funcionamento de cada tipo de nodo.

4.6.4 Nodo básico

Como já foi mencionado, o nodo básico é o elemento principal de distribuição da aplicação e funciona de forma semelhante aos objetos ativos de muitas linguagens de programação concorrente orientada a objeto. Os métodos da interface `XBaseNode` (TABELA 4.10) implementam essa característica.

TABELA 4.10 - Métodos da interface XBaseNode

método	descrição
`[get|set]CodeGenerator`	obtém/define o gerador de código
`startup`	inicia o objeto
`shutdown`	encerra o objeto
`run`	método chamado após o início da execução do objeto. É o método que contém o código com a lógica do usuário

Afora os métodos de controle do gerador de código, os outros métodos tornam o nodo básico muito semelhante ao objeto `Thread` da linguagem Java, onde existem os métodos `start` e `stop` que iniciam e encerram a execução da *thread*. Assim como na *thread*, o método `run` contém o corpo de execução do objeto.

Aos nodos básicos podem ser adicionados componentes e métodos. Os componentes são objetos com funções específicas que quando adicionados ao nodo básico são referenciados dentro dele, permitindo que ele tenha acesso a todas as suas funções. Os objetos do tipo método são representações de métodos comuns de qualquer classe Java. A diferença entre se digitar diretamente no código textual do objeto e inserir um objeto do tipo método está na possibilidade de se visualizar no próprio grafo os métodos que fazem parte do objeto.

Foram implementados quatro tipos de nodos básicos: o objeto distribuído comum, o objeto distribuído Voyager, o objeto RMI e o objeto serviço, explicados a seguir.

4.6.4.1 Objeto distribuído

O tipo de nodo básico mais simples é o objeto distribuído. Esse objeto não utiliza nenhum ambiente especial de execução distribuída e por isso não utiliza meios de comunicação nem de distribuição de alto nível (os motivos do não uso de um ambiente de distribuição em determinados casos foram apresentados na seção 2.5.1). A comunicação entre objetos desse tipo só acontece por meio de *sockets* e a distribuição deve ser feita de forma manual, com o usuário executando diretamente os objetos nas máquinas que desejar.

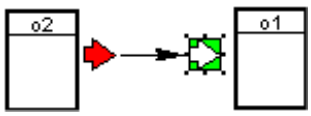


FIGURA 4.9 - Objeto distribuído normal numa aplicação cliente-servidor com *sockets*.

Um exemplo de utilização de um nodo básico é na implementação de uma aplicação cliente-servidor com *sockets*. A FIGURA 4.9 mostra uma aplicação formada pelo objeto `o2` que envia uma mensagem por *socket* ao objeto `o1`.

4.6.4.2 Serviço

O serviço é um tipo especial de nodo básico e implementa a interface `XServiceNode`, a qual estende `XBaseNode`. O serviço é utilizado em conjunto com outros objetos para a construção de uma aplicação cliente-servidor (vide seção 4.6.8.2.2). Nessas aplicações, quando uma requisição de serviço é aceita, um objeto do tipo serviço é criado para tratá-la.

TABELA 4.11 - Métodos da interface XServiceNode

método	descrição
`[get|set]ServicePort`	obtém/define a porta de serviço utilizada na comunicação com o cliente
`[get|set]ServiceData`	obtém/define o objeto que será compartilhado por todos os objetos criados pela mesma porta de conexão de serviço

A TABELA 4.11 apresenta os métodos adicionais dessa interface que são usados para o controle da comunicação com o cliente que requisitou o serviço. Para esta comunicação, o objeto serviço contém uma porta de serviço que é iniciada com o canal de comunicação ligado diretamente ao cliente e por onde trafegam os dados. O controle da porta de serviço é feito por meio do primeiro conjunto de métodos da tabela.

O segundo conjunto trata do compartilhamento de dados entre os serviços criados pela mesma porta de conexão. Esse espaço é representado por um objeto do tipo `ServiceData`, que contém as referências para todos os serviços criados. A razão de se colocar essa funcionalidade foi a de que ela é útil quando uma conexão necessita ter acesso aos dados de outras conexões criadas no mesmo serviço, como por exemplo num servidor de bate-papo. Nessa aplicação, quando uma mensagem deve ser enviada a todos os clientes ou a parte deles, o objeto que faz o envio deve conhecer o endereço de vários clientes e estas informações estão disponíveis apenas nos serviços criados para os clientes correspondentes.

4.6.4.3 Objeto Voyager

O objeto Voyager foi incluído entre os nodos básicos porque ele possui a maior parte das características desejáveis de um sistema de objetos distribuídos. Num ambiente distribuído Voyager é possível se fazer invocação remota de método, criação remota, coleta de lixo distribuída e outras atividades importantes em ambientes de objetos distribuídos.

A implementação desse nodo básico é relativamente simples. Ele é um nodo básico normal com um gerador de código específico para o ambiente Voyager e métodos `startup` e `shutdown` redefinidos. Esses métodos fazem o *login* e o *logout*, respectivamente, do objeto na máquina virtual Voyager.

4.6.4.4 Objeto RMI

A implementação é análoga ao do objeto Voyager. O objeto RMI foi implementado porque RMI é um sistema de objetos distribuídos mais disponível que o Voyager, pois já faz parte do JDK da Sun. A desvantagem é que ele não é tão poderoso quanto Voyager, apresentando deficiências importantes como a impossibilidade de se criar objetos remotos. Mesmo assim, cumpre o seu papel na invocação remota de métodos, que é o seu principal objetivo.

4.6.5 Nodo de interface

Os únicos métodos que um nodo de interface deve implementar são os que gerenciam o gerador de código (TABELA 4.12). O gerador de código do nodo de interface é uma simplificação do gerador do nodo básico, pois o código da interface contém apenas a declaração da própria interface, dos métodos e dos pacotes importados.

TABELA 4.12 - Métodos da interface XInterfaceNode

método	descrição
`[get|set]CodeGenerator`	obtém/define o gerador de código utilizado

Considerando os nodos de interface implementados (`XVoyagerInterface`, `XRMIInterface` e `XJavaInterface`), as únicas diferenças estão no gerador de código e na representação visual. O gerador de código de cada um é utilizado para criar a declaração da interface e responder à inserção de métodos. Além disso, o gerador de código da interface é chamado quando um nodo básico implementa `XInterfaceNode`, pois as declarações correspondentes dos métodos no nodo básico dependerão de como elas são feitas no nodo de interface em questão.

4.6.6 Nodo terminal

Como foi explicado na seção 3.4.6, o objetivo do nodo terminal é formar uma biblioteca de componentes da ferramenta. Deseja-se com isso que o nodo terminal esteja para esta ferramenta como os componentes JavaBeans estão para o ambiente Java. Nos dois casos a principal finalidade é oferecer meios de reutilização do código já desenvolvido, permitindo que um nodo básico composto por diversos componentes, criado para uma aplicação específica, esteja disponível a outras aplicações de forma simples.

Para permitir que um nodo básico seja integrado transparentemente a uma aplicação, mantendo os componentes, a aparência visual e todas as outras características que tinha ao ser construído pela primeira vez, informações adicionais devem ser dadas à ferramenta quando ela tentar importar um desses componentes. Para este fim existe a interface `XTerminalNode`, cujos métodos estão na TABELA 4.13.

TABELA 4.13 - Métodos da interface XTerminalNode

método	descrição
`[get|set]baseNode Class`	obtém/define a classe do nodo básico representado por este nodo terminal
`getTerminalInfo`	retorna as informações do objeto contido pelo nodo terminal

TABELA 4.14 - Métodos da interface XTerminalInfo

método	descrição
`getComponent Descriptors`	retorna descritores do tipo `XComponentDescriptor` contendo informações sobre os componentes que fazem parte do nodo básico representado por este nodo terminal
`getMethod Descriptors`	retorna descritores do tipo `XMethodDescriptor` com as informações a respeito dos métodos inseridos no nodo básico representado por este nodo terminal
`getIcon`	retorna um ícone para ser colocado na paleta de componentes

O segundo método da TABELA 4.13 é análogo ao método `getBeanInfo` da interface `BeanInfo` do pacote java.beans. Este método retorna informações para que uma ferramenta processe os componentes, enquanto o método `getTerminalInfo` retorna informações para tratamento dos nodos terminais. Os dados que devem ser retornados são principalmente os componentes contidos pelo nodo básico, seus valores de propriedades e seus eventos. Com estas informações, a ferramenta é capaz de tomar o nodo terminal e reconstruí-lo de acordo com o seu estado original.

As informações retornadas pelo método `getTerminalInfo` estão organizadas num objeto que implementa a interface `XTerminalInfo`, cuja estrutura está na TABELA 4.14. Um exemplo torna a compreensão dessa interface bem mais fácil (FIGURA 4.10).

```
public class TesteTerminalInfo extends XDefaultTerminalInfo {
  public XComponentDescriptor[] getComponentDescriptors() {
    XComponentDescriptor componentDescriptor = new
       XComponentDescriptor(Node.BaseNode, "vpj.base.XVoyagerObject",
                            "Teste", 0, 0, 36, 50);

    XComponentDescriptor componentDescriptor_0 = new
       XComponentDescriptor(Node.ProxyNode, "vpj.proxy.XVoyagerProxy",
                            "c1", 38, 0, 16, 16);
    componentDescriptor.addChild(componentDescriptor_0);

    XComponentDescriptor componentDescriptor_1 = new
       XComponentDescriptor(Node.ProxyNode, "vpj.proxy.XRMIProxy",
                            "c2", 38, 30, 16, 16);
    componentDescriptor.addChild(componentDescriptor_1);

    XComponentDescriptor[] componentDescriptors = new
       XComponentDescriptor[1];
    componentDescriptors[0] = componentDescriptor;
    return componentDescriptors;
  } // getComponentDescriptors

} // TesteTerminalInfo
```

FIGURA 4.10 - Exemplo de código XTerminalInfo para um objeto.

Este código é gerado pela ferramenta quando o usuário comanda a transformação de nodo básico para nodo terminal. Neste exemplo, inicialmente há um nodo básico da classe `vpj.base.XVoyagerObject` e de nome `Teste`, contendo dois componentes: `c1` da classe `vpj.proxy.XVoyagerProxy` e `c2` da classe `vpj.proxy.XRMIProxy` (os valores numéricos colocados ao lado são as posições físicas relativas ao objeto que o contém e as suas dimensões, que são usadas na apresentação visual).

Quando o nodo terminal da classe `Teste` é inserido num grafo, a ferramenta chama o método `getTerminalInfo` do nodo terminal que, por ***default***, procura a classe `TesteTerminalInfo` e retorna as informações chamando o método `getComponentDescriptors` desse objeto.

Portanto, é possível se construir componentes para esta ferramenta transformando nodos básicos em nodos terminais. Dessa maneira foram construídos dois nodos terminais: os componentes Lock e Global. Embora o modelo de programação da ferramenta não exija que determinados tipos de componentes estejam disponíveis, o fato desta ferramenta se destinar à programação com objetos distribuídos faz com que alguns tipos de componentes sejam disponibilizados com este objetivo. A programação com objetos distribuídos nessa ferramenta exige a definição de como irão funcionar os

elementos de distribuição, como será a comunicação entre eles e como outras características desejáveis de um programa concorrente, como um sistema de *lock* global, serão usados. A seguir então serão descritas as funcionalidades e características principais dos componentes Lock e Global.

4.6.6.1 Lock

O *lock* é o objeto utilizado para o tratamento de seções críticas de código. Como é possível se ter elementos globais compartilhados como arquivos, por exemplo, é necessária a existência de um tipo especial de sincronização global. Existem vários mecanismos de sincronização que podem ser usados nessa tarefa [TAN97], mas o *lock*, por sua simplicidade, foi escolhido para ser usado nesta ferramenta.

Há dois tipos de componentes envolvidos na implementação do *lock*: o cliente e o servidor do *lock*. Uma aplicação pode ter mais de um servidor, que se responsabiliza pelo controle de diferentes clientes. Os clientes fazem as chamadas de requisição à liberação de um recurso específico (o *lock*) e o servidor, com base no estado local, concede a requisição ou a coloca numa fila para concessão posterior.

Existem duas classes de *lock*: o *lock* de leitura e o *lock* de escrita. Os *locks* de leitura podem conviver sem problemas entre si, ou seja, vários *locks* de leitura podem ser concedidos sobre um objeto sem que surjam inconsistências. Os *locks* de escrita, entretanto, devem ser exclusivos. Apenas um objeto pode estar numa seção crítica com permissão de escrita ao mesmo tempo.

Caso a aplicação do usuário necessite, é também possível se ter *locks* de escrita e de leitura sujos, caso em que é liberada a simultaneidade de leitura e de escrita. Nesse caso, entretanto, não se garante a integridade dos dados lidos. A vantagem que se tem é o aumento da concorrência, o que pode ser mais importante do que a correção dos dados para algumas aplicações.

A parte cliente do objeto *lock* segue o mesmo modelo de eventos utilizado nos outros objetos desta ferramenta. As requisições de *lock* de escrita e de leitura são sempre assíncronas. Quando a operação é concedida, um evento é gerado e retornado para o objeto que o requisitou. Isso permite que o objeto não fique bloqueado apenas para esperar a liberação do *lock*. Num caso extremo, ele poderia ficar bloqueado eternamente sem que o *lock* fosse liberado. Com esse esquema de eventos, ele faz a requisição e parte imediatamente para a realização de outra tarefa. Quanto o evento de que o *lock* foi concedido é gerado, ele pode realizar o processamento sobre a seção crítica.

4.6.6.2 Global

Determinadas aplicações se tornam mais fáceis de serem implementadas quando é possível se ter uma área de dados comum entre diferentes objetos distribuídos. Para isso, esta ferramenta apresenta um tipo de objeto, o Global, que permite que diferentes objetos compartilhem uma estrutura de dados comum.

Quanto à implementação, assim como no objeto do tipo *lock*, há uma parte cliente e uma parte servidora. A parte servidora armazena um estado local que está acessível aos objetos clientes através de uma interface conhecida. Esse objeto controla o acesso aos dados e os disponibiliza através da rede, para aqueles que os requisitarem, por um meio de comunicação que pode ser chamada remota de método ou mensagem explícita.

4.6.7 Nodo virtual

A interface `XVirtualNode` não possui nenhum método, tornando a implementação de um nodo virtual bastante livre. Foram implementados os seguintes nodos virtuais:

- **servidor virtual**: representa uma máquina com nome e número de porta para receber conexões TCP;
- **arquivo**: representa um arquivo num sistema de arquivos distribuído disponível a todos os objetos do grafo;
- **impressora**: representação lógica do recurso físico impressora;
- **objeto RMI**: representa uma implementação externa de um objeto RMI;
- **objeto Voyager**: representa uma implementação externa de um objeto Voyager.

Os nodos virtuais são sempre interfaces de acesso a recursos externos, o que os torna componentes bem simples, já que a funcionalidade em si do objeto é implementada fora da aplicação.

4.6.8 Componente

Um componente não precisa implementar nenhuma interface para ser considerado como tal pelo ambiente de programação. Portanto, quando o objeto não implementa nenhuma das interfaces citadas, é automaticamente considerado como um objeto do tipo componente. Isso permite que qualquer componente JavaBean desenvolvido para e em outras ferramentas possa ser adicionado a esta ferramenta.

Os componentes implementados são os mais diversos, mas há duas categorias principais: os componentes utilizados nas interfaces gráficas das aplicações e as portas de comunicação. Estas últimas são muito importantes, pois implementam a comunicação por *sockets* entre os objetos que não têm acesso a chamadas remotas de métodos.

4.6.8.1 Interface gráfica

O objetivo desta ferramenta não é o desenvolvimento de interfaces gráficas. Para isso existem ferramentas mais apropriadas como o JBuilder e o IBM VisualAge que possuem uma grande quantidade de recursos para automatizar essa tarefa. Entretanto, para que as aplicações desenvolvidas nessa ferramenta não se limitem a meios textuais de entrada e apresentação de dados, foram desenvolvidos alguns componentes com a função de realizar essas funções graficamente. Inicialmente são oferecidos dois tipos de componentes:

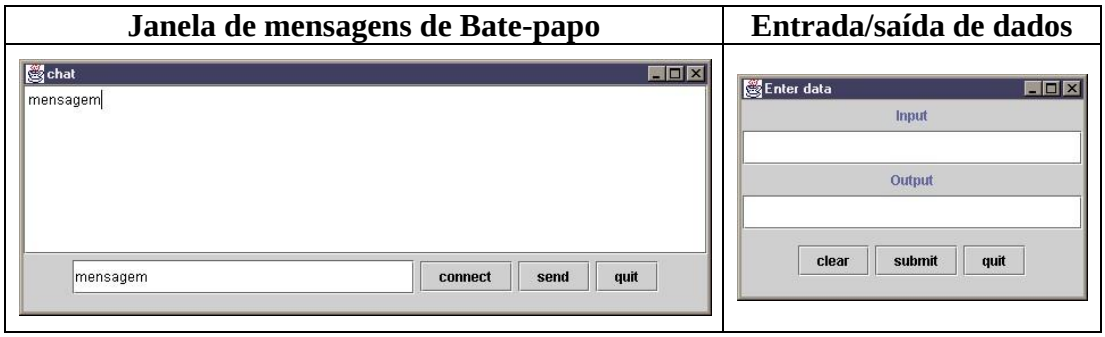


FIGURA 4.11 - Componentes de interface gráfica.

- **bate-papo**: interface gráfica simples para uma aplicação de bate-papo;
- **entrada e saída de dados**: uma janela contendo três botões, um campo para entrada de dados e outro para saída.

A forma de se integrar esses objetos com a aplicação é a mesma dos outros componentes, ou seja, através das propriedades, métodos e eventos. Como foi melhor discutido na seção 3.3.5, assim como qualquer outro componente, o componente de interface gráfica pode ser desenvolvido externamente e adicionado à ferramenta. Basta para isso seguir o padrão JavaBeans de desenvolvimento de componentes.

4.6.8.2 Portas de comunicação

Embora não faça parte da concepção dos objetos distribuídos, o suporte desta ferramenta à comunicação por meio de *sockets* é importante para que dois objetos localizados em máquinas diferentes possam se comunicar ainda que um sistema de gerenciamento de objetos distribuídos não esteja presente. Não somente por isto, mas também porque este tipo de comunicação facilita a interconexão entre sistemas programados em diferentes plataformas de *hardware*, sistema operacional e linguagem de programação. Esta questão está relacionada com a portabilidade, que foi um dos principais objetivos estabelecidos no início deste trabalho.

As vantagens das portas de comunicação não estão apenas na sua implementação, mas também no seu modelo. Por exemplo, em nível de aplicação, não é necessário que se conheça o objeto remoto destino da mensagem, pois a porta é uma referência local ao transmissor. Assim, se o destino da mensagem for alterado, o código fonte do transmissor não precisa sofrer nenhuma modificação, exceto na parte que diz respeito à criação e configuração da porta, onde são informados os parâmetros do destino do canal.

Com relação à linguagem visual utilizada nesta ferramenta, as portas de comunicação são objetos do tipo componente que, por isso, devem estar atrelados a um nodo básico para que possam ser usados. Desse modo, a comunicação entre a porta e o nodo básico se dá através de eventos. Quando algo de importante acontece na porta, como por exemplo a chegada de uma mensagem, um evento é gerado para o objeto que detém essa porta.

As portas permitem a utilização de dois formatos de dados nas mensagens. É possível se transmitir dados em formato básico (cadeia de caracteres - utilizado na maior parte da comunicação feita por *sockets* nos computadores em geral) e em forma de objetos serializáveis de Java. Essa segunda opção pode facilitar bastante a implementação de determinados programas, quando o cliente e o servidor foram desenvolvidos em Java e assim suportam a serialização de objetos.

Foram implementados componentes com a função de realizar a comunicação entre objetos através do protocolo TCP. Não foram implementados componentes apenas com o fim exclusivo de comunicação, mas também outros com características especiais para a criação de aplicações cliente-servidor. Caso o usuário não esteja satisfeito com as opções oferecidas e necessitar de portas de comunicação com um funcionamento diferente, como por exemplo uma porta UDP, basta que ele implemente esse componente e o adicione à ferramenta.

A seguir será descrito o funcionamento das portas de comunicação por *sockets* implementadas nesta ferramenta.

4.6.8.2.1 Porta de entrada, porta de saída e porta de entrada e saída

Para comunicação simples, com envio e recebimento de mensagens, foram implementadas as seguintes portas de comunicação, cujos detalhes estão na TABELA 4.15:

- **InputPort**: porta de entrada, para recebimento de dados;
- **OutputPort**: porta de saída, para envio de dados;
- **InputOutputPort**: porta para envio e recebimento de dados.

Apesar de que seja possível se utilizar a porta de entrada e saída em qualquer tipo de comunicação, houve-se por bem separá-la e criar também as portas que tenham cada uma somente a funcionalidade de envio e de recebimento, permitindo que os programas visuais ficassem mais representativos. Por exemplo, num relacionamento entre duas portas de entrada e saída é mais difícil se conhecer quem é o receptor e quem é o transmissor. Quando se utiliza uma porta de entrada e outra porta de saída, torna-se claro quem está enviando e quem está recebendo a mensagem.

TABELA 4.15 - Portas de comunicação simples.

porta	propriedades	métodos	eventos

`InputPort`	`localHost`
	`localPort`
	`protocol`
		`open`
		`close`
		`receive`
		`receiveAsync`
			`onCreate`
			`onClose`
			`onReceive`
			`onError`
			`onRemoteDisconnect`

`OutputPort`	`remoteHost`
	`remotePort`
	`protocol`
		`open`
		`close`
		`send`
		`sendSync`
			`onCreate`
			`onClose`
			`onSendComplete`
			`onError`
			`onRemoteDisconnect`

`InputOutputPort`	`remoteHost`
	`localHost`
	`remotePort`
	`localPort`
	`protocol`
		`open`
		`close`
		`receive`
		`receiveAsync`
		`send`
		`sendSync`
			`onCreate`
			`onClose`
			`onReceive`
			`onSendComplete`
			`onError`
			`onRemoteDisconnect`

As portas de comunicação são utilizadas sempre aos pares, uma no transmissor e outra no receptor. Para ilustrar esse comportamento e o seu funcionamento, será descrita a estrutura da porta de entrada e de saída na FIGURA 4.12 e o seu funcionamento a seguir. O funcionamento das outras é semelhante, diferindo apenas pela impossibilidade de enviar (na porta de entrada) ou de receber (na porta de saída) dados.

Quando a porta de entrada e saída é criada, são criados dois objetos adicionais: o notificador de entrada e o notificador de saída. Esses objetos são *threads* que tratam exclusivamente da comunicação, controlando os *sockets* de envio e recebimento de dados.

O notificador de entrada controla um *socket* servidor que fica esperando dados. Quando uma mensagem chega, ele a coloca na fila de recebimento de mensagens e gera um evento `onReceive` informando que uma mensagem chegou. Esse evento é transferido para o nodo básico que contém a porta, de maneira que ele possa ler a mensagem através do método `receive` da porta. Essa chamada faz com que a mensagem seja retirada da fila de recebimento e repassada ao nodo básico.

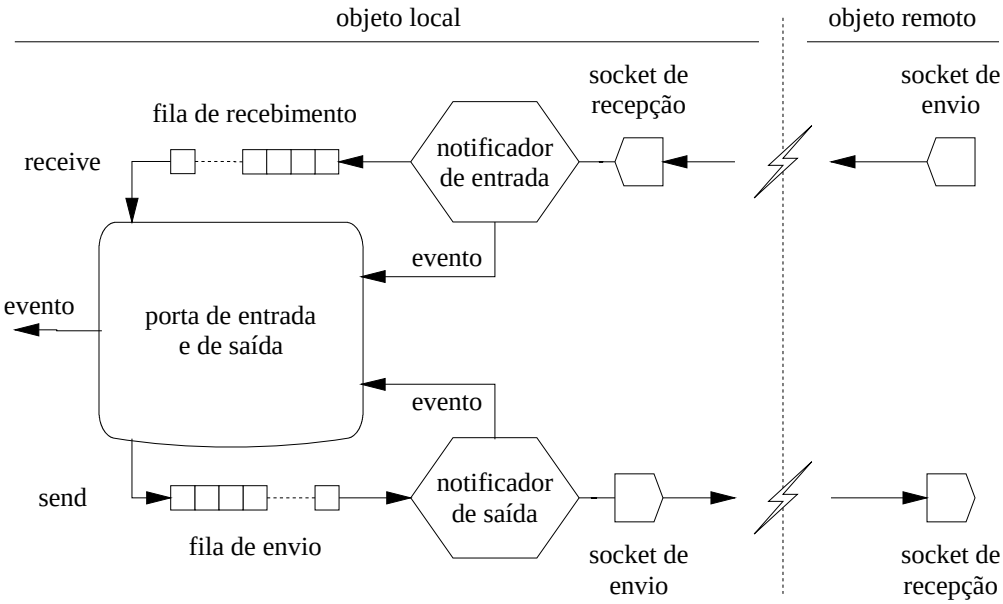


FIGURA 4.12 - Estrutura do funcionamento da porta de entrada e saída.

O funcionamento do notificador de saída é semelhante. Ele é uma *thread* que controla um *socket* cliente e monitora continuamente a fila de envio de dados da porta. Quando o nodo básico executa uma chamada `send` na porta, a mensagem é colocada na fila de envio. O notificador retira a mensagem da fila e a envia ao objeto remoto através do *socket*. Quando o envio da mensagem é completado, o notificador gera o evento `onSendComplete` que é repassado ao nodo básico que contém a porta.

A implementação do envio e recebimento das mensagens através dos notificadores evita que o nodo básico que contém a porta fique bloqueado enquanto as operações de rede são efetuadas. Numa chamada *send* tradicional, o tempo total de execução envolve a transferência completa e síncrona dos dados, de maneira que a chamada é liberada somente quando o *socket* remoto recebeu toda a mensagem. A implementação das portas de comunicação faz com que o tempo da chamada *send* seja somente o tempo de colocar a mensagem na fila. A transferência bloqueante é feita pelo notificador de saída, que avisa a porta através do evento `onSendComplete` quando a mensagem foi enviada.

Há ainda outros métodos nessa porta que fazem a transferência de dados de uma maneira diferente. O método `sendSync` realiza uma transferência direta de dados, sem passar pela fila. Neste caso, o objeto fica bloqueado até que toda a mensagem seja transferida. O método `receiveAsync` realiza uma chamada assíncrona de recebimento. Mesmo que não haja nenhuma mensagem esperando, o objeto não fica bloqueado.

Os notificadores também fazem o controle de erros na comunicação. Se algo acontece de errado na transferência de dados, é gerado o evento `onError`. Caso a conexão seja desfeita remotamente, é gerado o evento `onRemoteDisconnect`.

4.6.8.2.2 Porta de chamada de serviço, porta de conexão de serviço e porta de serviço

As portas apresentadas na seção anterior são utilizadas para a troca simples de mensagens. Para dar um suporte de mais alto nível a aplicações cliente-servidor, foi implementado um conjunto de portas que em geral são utilizadas simultaneamente:

- **CallPort:** porta para conexão a um serviço remoto;
- **ServiceConnectionPort:** porta que gerencia a chegada de pedidos de conexão de portas de chamada de serviço e cria os objetos que tratarão o serviço;
- **ServicePort**: porta utilizada na comunicação de um objeto de tratamento do serviço com a porta de chamada de serviço.

As características de cada porta são descritas na TABELA 4.16 e o relacionamento que têm entre si é apresentado de forma esquemática na FIGURA 4.13.

TABELA 4.16 - Portas de comunicação em aplicações cliente-servidor.

porta	propriedades	métodos	eventos

`CallPort`	`remoteHost`
	`remotePort`
	`protocol`
		`open`
		`close`
		`receive`
		`receiveAsync`
		`send`
		`sendSync`
			`onCreate`
			`onClose`
			`onConnectionRequest`
			`onConnectionAccept`
			`onConnectionRefuse`
			`onReceive`
			`onSendComplete`
			`onError`
			`onRemoteDisconnect`

`ServiceConnectionPort`	`localHost`
	`localPort`
	`serviceClass`
	`protocol`
		`open`
		`acccept`
		`refuse`
		`close`
			`onCreate`
			`onClose`
			`onConnectionRequest`
			`onError`
			`onRemoteDisconnect`

`ServicePort`	`protocol`
	`clientSocket`
		`open`
		`close`
		`receive`
		`receiveAsync`
		`send`
		`sendSync`
			`onCreate`
			`onClose`
			`onReceive`
			`onSendComplete`
			`onError`
			`onRemoteDisconnect`

A propriedade `clientSocket` da porta `ServicePort` é o objeto passado da porta de conexão à porta de serviço do nodo básico de serviço no momento de estabelecimento da conexão. É através deste *socket* que o servidor irá se comunicar com o cliente.

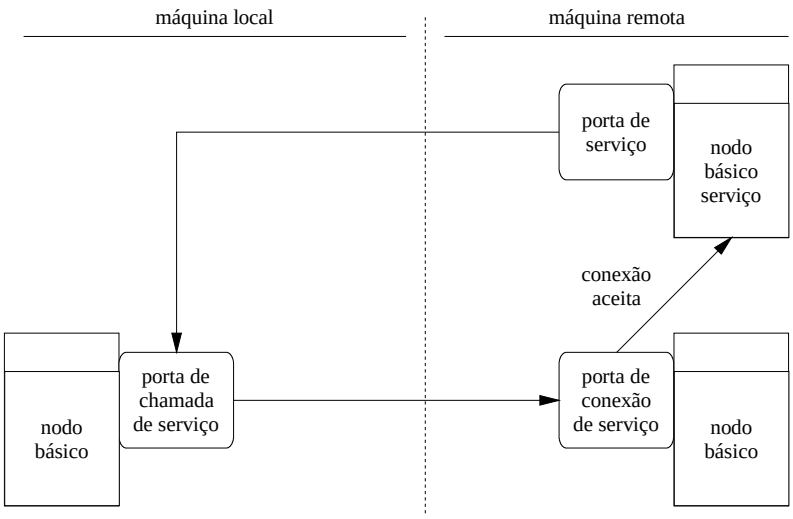


FIGURA 4.13 - Portas numa aplicação cliente-servidor com *sockets*.

O funcionamento interno dessas portas é semelhante ao das portas apresentadas na seção anterior. A porta de chamada de serviço e a porta de serviço possuem notificadores de entrada e de saída, pois essas portas podem enviar e receber mensagens. A porta de conexão de serviço possui apenas o notificador de entrada que fica esperando por novas conexões.

A dinâmica das conexões flui da seguinte maneira: a porta de chamada de serviço realiza um pedido de conexão à porta de conexão de serviço, que por sua vez pode aceitar ou rejeitar o pedido. Caso o pedido seja aceito, ela cria um novo objeto do tipo serviço, que já possui uma porta de serviço embutida (a classe desse objeto é especificada na propriedade `serviceClass` da porta de conexão de serviço). Quando esse objeto é criado, a conexão feita com o objeto cliente é redirecionada para a porta de serviço, fazendo com que a partir desse momento a ligação entre a porta de chamada de serviço e a porta de conexão seja desfeita, e esta passe a esperar por novas conexões.

É desse modo que são implementadas as aplicações cliente-servidor com portas *sockets* nessa ferramenta. Mais detalhes sobre essas aplicações, com exemplos, podem ser encontrados nos arquivos de distribuição da ferramenta. Lá está um exemplo de uma aplicação de bate-papo implementada com estes componentes.

4.6.9 Componente procurador

O componente procurador deve ter os métodos necessários para que ele possa dar ao objeto uma visão local do objeto remoto. Isto é conseguido por meio do método `returnRemoteObject` (TABELA 4.17), que retorna uma referência do tipo da interface à qual o componente procurador está conectado. É por causa da necessidade desse tipo de referência que o componente procurador deve sempre se conectar à interface.

TABELA 4.17 - Método da interface XProxyNode

método	**descrição**
`returnRemoteObject`	retorna uma referência local para o objeto instanciado remotamente

Dois tipos de componentes procuradores foram implementados: o `XVoyagerProxy` e o `XRMIProxy`. Na implementação de `returnRemoteObject` nesses dois componentes, a referência é retornada através de uma consulta ao servidor de nomes, passando-se como parâmetro o nome do objeto remoto. A referência retornada é, na verdade, uma referência a um objeto local que implementa a transferência das mensagens entre o ambiente local e o ambiente remoto. Esse objeto local é fornecido pelo ambiente de execução distribuída, de maneira que o programador não precisa se preocupar com essa tarefa.

4.6.10 Componente para criação dinâmica

A implementação deste tipo de componente compreende dois tipos de métodos principais. Eles são necessários porque este componente, para exercer a sua função de criar objetos remotamente, necessita conhecer o nome da classe à qual o objeto remoto pertence e o instante em que ele deve ser criado. Os primeiros métodos (`[get|set]remoteClassName`) servem para definir o nome da classe remota. O método `createRemoteObject` cria efetivamente o objeto remoto e deve ser chamado explicitamente quando a criação for desejada pelo usuário.

TABELA 4.18 - Métodos da interface XDynamicNode

método	descrição
`[get|set]remoteClassName`	gerencia a classe do objeto remoto
`createRemoteObject`	instancia o objeto remotamente

Após criar um objeto, este componente se comporta como um componente procurador, retornando uma referência local para o objeto remoto. Dessa forma, não é necessário que se utilize além de um componente de criação dinâmica um componente procurador para tratar as invocações remotas subseqüentes.

Dois tipos de componentes de criação dinâmica foram desenvolvidos: `XVoyagerDynamic` e `XRMIDynamic`. Como RMI não possui a possibilidade de criar dinamicamente um objeto remoto, na implementação desta função é utilizado o ambiente Voyager. As invocações remotas seguintes, entretanto, são feitas com RMI.

Uma extensão possível a este componente é o acréscimo de métodos para aumentar a flexibilidade. Por exemplo, poderia ser adicionado um método que permitisse a criação de matrizes de objetos ou ainda de objetos replicados. Esses métodos, entretanto, foram deixados como sugestões para trabalhos futuros.

4.6.11 Método

O nodo método deve apenas implementar a interface `XMethodNode`, permitindo que ele seja inserido em nodos de interface e em nodos básicos. O objeto implementado foi o `XJavaMethod`.

4.6.12 Nodo de interface de grafo

Este objeto deve implementar a interface `XGraphInterfaceNode`, que não possui nenhum método. O único objeto implementado foi `XGraphInterface`.

4.6.13 Chamada a grafo

O nodo de chamada a grafo possui uma implementação mais complexa do que a dos demais. Embora possua apenas dois métodos que gerenciam o nome do arquivo do grafo que ele representa, sua complexidade é grande devido à tarefa de carga dos grafos e apresentação dos objetos.

TABELA 4.19 - Métodos da interface XGraphCallNode

método	descrição
`[get|set]graphName`	gerencia o nome do arquivo que contém o grafo representado na chamada a grafo

O arquivo de grafo usado por este nodo é o mesmo usado para abrir um grafo no editor de grafos. O arquivo com a representação de nodos e arcos é lido do disco, sendo pesquisados os arcos de interface de grafo. Os objetos conectados a esses tipos de arcos são parte da interface do grafo, fazendo com que então sejam apresentados no editor.

Para que os objetos participantes deste nodo sejam diferenciados dos demais, quando um relacionamento é criado com um deles, ao nome indicado no relacionamento é adicionado o nome do grafo. Mais detalhes sobre isso serão vistos na seção 4.8 que trata especificamente de grafos e subgrafos.

4.6.14 Nodo de documentação

Um nodo de documentação deve apenas implementar a interface `XDocumentationNode`, que não possui métodos. As classes implementadas foram `XLabel` e `XImage`, que permitem que textos e imagens, respectivamente, sejam apresentados no grafo.

4.7 Relacionamentos

Até este momento foram descritas as implementações dos objetos colocados de forma independente uns dos outros. A outra característica importante da linguagem visual, o relacionamento, terá a sua implementação descrita a partir de agora.

Via de regra, os relacionamentos podem ser visualizados diretamente no grafo através de arcos. Para conhecer os detalhes do relacionamento, o usuário pode selecionar o arco correspondente e visualizar as suas propriedades, que serão específicas ao tipo de relacionamento formado.

4.7.1 Normal

A implementação desse relacionamento se dá pelo mapeamento de propriedades entre o nodo fonte e o nodo destino. Ao tentar se inserir um arco normal, é apresentada uma janela contendo as propriedades dos objetos envolvidos para que o usuário estabeleça os mapeamentos das propriedades e assim concretize o relacionamento.

A FIGURA 4.14 mostra a janela que é apresentada ao se tentar criar o arco normal da FIGURA 4.9. A parte de cima da janela (*Current mappings*) mostra os mapeamentos já construídos. Eles significam que, durante o projeto da aplicação, os valores das propriedades `remoteHost` no nodo fonte e `host` no nodo destino serão os mesmos, o que vale também para o mapeamento entre as propriedades `remotePort` e `port`. Assim, durante a construção do grafo, se o valor de `host` no nodo destino for alterado para *poncho.inf.ufrgs.br*, `remoteHost` no nodo fonte receberá este valor.

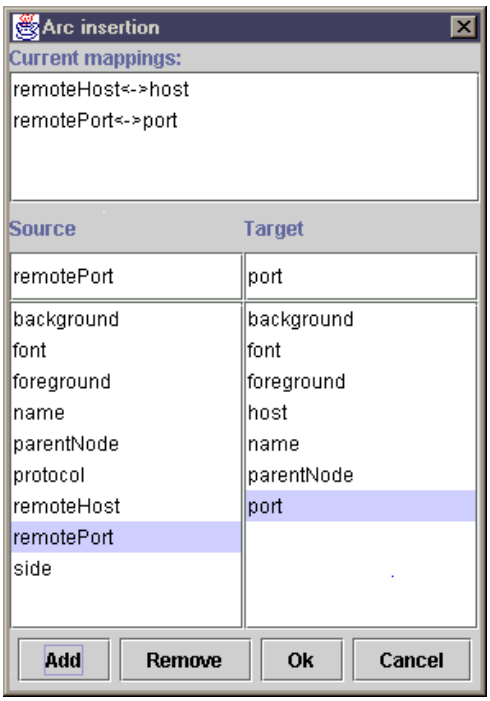


FIGURA 4.14 - Janela de mapeamento de propriedades.

As alterações nos valores das propriedades funcionam da mesma maneira como se as propriedades estivessem sendo editadas interativamente pelo usuário. Dessa forma, além de efetivamente mudar o valor da propriedade, representações e códigos fontes gerados podem ser modificados durante a criação desse tipo de relacionamento.

4.7.1.1 Mapeamentos predefinidos

Para evitar que o usuário tenha que especificar explicitamente os mapeamentos de propriedades toda vez que inserir um arco normal, existe a interface `XDefaultPropertyMapping` (TABELA 4.20) que relaciona propriedades entre objetos.

TABELA 4.20 - Métodos da interface XDefaultPropertyMapping

método	descrição
`getDefaultPropertyMappingsFrom`	retorna um vetor contendo os mapeamentos de propriedades quando o objeto que implementa esta interface é origem do arco
`getDefaultPropertyMappingsTo`	similar ao anterior, retorna os mapeamentos quando o objeto é destino do arco

Quando um arco normal é inserido entre dois objetos, a ferramenta verifica através dos métodos dessa interface se há propriedades que podem ser mapeadas automaticamente. Por exemplo, quando um arco é inserido entre um componente `XOutputPort` e um `XInputPort`, algumas propriedades (FIGURA 4.15) são imediatamente combinadas e a janela de inserção de arco não é apresentada, pois essas classes implementam a interface `XDefaultPropertyMapping`.

XOutputPort	XInputPort
`remotePort`	`port`
`remoteHost`	`host`
`protocol`	`protocol`

FIGURA 4.15 - Exemplo de mapeamento predefinido.

Além de facilitar o desenvolvimento, em muitos casos essa possibilidade é essencial, pois o usuário pode nem saber quais propriedades ele deve combinar para que o relacionamento funcione na prática. Por isso, boa parte dos componentes desenvolvidos nessa ferramenta já possui os mapeamentos definidos, de maneira que o usuário não terá trabalho adicional quando inserir um arco normal entre objetos que costumam fazer parte de relacionamentos.

A implementação da interface `XDefaultPropertyMapping` não é obrigatória. Caso essa implementação não aconteça, a criação de relacionamentos entre os componentes tradicionais continua sendo através da definição explícita dos mapeamentos de propriedades com intervenção do usuário.

4.7.2 Implementação de interface

A forma como é implementado este relacionamento é a mesma da programação textual em Java. Ou seja, quando um nodo básico implementa a interface, à declaração da sua classe é acrescentada a declaração da implementação da interface.

O exemplo apresentado a seguir (FIGURA 4.16, FIGURA 4.17 e FIGURA 4.18) mostra, respectivamente, a representação visual da implementação da interface, o código gerado para o nodo de interface e o código do nodo básico que implementa essa interface.

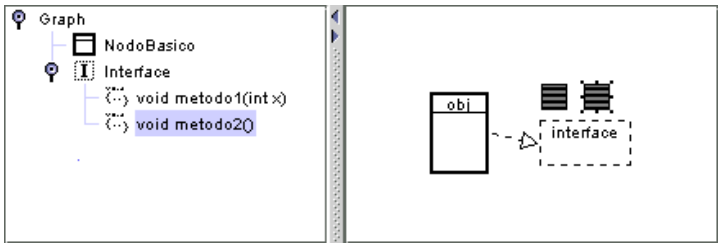


FIGURA 4.16 - Implementação de interface.

```
public interface Interface {
    void metodo1(int x);
    void metodo2();
}
```

FIGURA 4.17 - Código do nodo de interface.

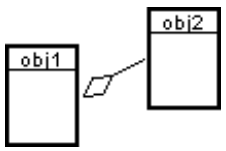
```
public class NodoBasico extends vpj.base.XDistributedObject implements
Interface {
    public NodoBasico() {
        super();
        this.setName("obj");
    }

    // method defined in the interface <Interface>
    public void metodo1(int x) {
    }

    // method defined in the interface <Interface>
    public void metodo2() {
    }
}
```

FIGURA 4.18 - Código do nodo básico.

Parte do código foi omitida para evitar a apresentação de detalhes desnecessários. A FIGURA 4.16 - mostra a representação gráfica do nodo de interface contendo dois métodos. Quando o nodo básico é conectado ao de interface, a declaração do nodo básico é alterada automaticamente e o esqueleto da implementação dos métodos definidos na interface também é adicionado ao código (FIGURA 4.18).

4.7.3 Compartilhamento de memória

A implementação do compartilhamento de memória é bem simples. Ao objeto fonte do relacionamento é adicionada a declaração do objeto destino, de forma que este último é instanciado no primeiro, como mostra o exemplo da FIGURA 4.19.




```
public class Obj1 extends XDistributedObject {
    obj2 obj2_obj;

    public obj1() {
        super();
        obj2_obj = new obj2();
        obj2_obj.setParent(this);
    }
}
```

FIGURA 4.19 - Código mostrando a declaração do objeto compartilhado.

No exemplo acima, o nodo básico `obj1` possui uma referência ao nodo básico `obj2`, pois foi ele quem o criou. Através dela, `obj1` tem acesso a todos os atributos e métodos públicos de `obj2`. Para que `obj2` também possa ter acesso aos métodos de `obj1`, ele recebe a referência por meio do método `setParent` e atualiza a propriedade "nodo pai" (propriedade do nodo básico vista na seção 3.4.3)

O relacionamento criado é de composição [FOW97] e pode acontecer em vários níveis. Assim, um nodo pode compor outros, que por sua vez também pode fazer o mesmo e assim por diante.

4.7.4 Interface de objeto remoto

Quando se deseja ter acesso a um objeto remoto ou criá-lo, é necessário que antes se especifique a interface por ele implementada. Isto é feito através da conexão de um componente procurador ou de criação dinâmica a um nodo de interface, o que caracteriza o relacionamento interface de objeto remoto (vide seção 3.4.18).

Assim, quando um relacionamento desses é estabelecido, tudo o que estiver relacionado com o componente e que envolver a interface é adicionado ou alterado, como por exemplo a criação da declaração do objeto remoto. Além disso, entre outras coisas, adiciona-se ao código do nodo básico correspondente um método que irá automatizar a busca da referência remota e a criação do objeto remoto (FIGURA 4.20 e FIGURA 4.21, respectivamente). Esse código não é indispensável, visto que o usuário poderia usar os métodos dos componentes diretamente. Mas como a finalidade dessa ferramenta é justamente diminuir o trabalho do programador, essas facilidades foram implementadas.

4.7.5 Referência a objeto remoto

Este relacionamento se estabelece sempre entre um componente procurador e um nodo básico. O arco criado é do tipo normal, implementado através de mapeamento de propriedades. Como tanto o componente procurador como o nodo básico possuem propriedades específicas de seus tipos de objetos (`name` e `remoteName`, respectivamente), esse relacionamento passa a ser designado como um caso especial do arco normal. Assim, o relacionamento é implementado através de mapeamento de propriedades, o que significa que o uso de mapeamentos predefinidos também é aplicável a este caso.

Vale sempre lembrar que o mapeamento de propriedades é definido em tempo de projeto, ou seja, se o valor da propriedade mapeada alterar em tempo de execução, a menos que um tratamento especial tenha sido incluído pelo usuário, nada garante que a propriedade correspondente no outro nodo também vá ser alterada. Por exemplo, se o valor da propriedade `host` for alterada no nodo básico destino, o componente procurador não será notificado da alteração.

```
InterfaceRemota pr_InterfaceRemota; // componente procurador

pr_InterfaceRemota = (InterfaceRemota) pr_returnRemoteObject();
```

FIGURA 4.20 - Método que retorna a referência local ao objeto remoto.

A FIGURA 4.20 mostra o método criado no nodo básico que automatiza a busca da referência do objeto remoto com base nas propriedades definidas para cada objeto. Basta que o usuário invoque este método e utilize a referência retornada para realizar uma chamada remota, da mesma forma que o faz com chamadas locais.

A busca da referência remota ao objeto se dá por meio de uma consulta ao servidor de nomes. Nos dois objetos implementados para este fim, a consulta se dá por meio dos métodos expostos na TABELA 4.21.

TABELA 4.21 - Busca no servidor de nomes da referência ao objeto remoto.

ambiente	método
RMI	`Naming.lookup("//host/remoteName")`
Voyager	`Namespace.lookup("//host/remoteName")`

O usuário não precisa conhecer os métodos da tabela acima. Ele apenas chama o método `returnRemoteObject` e o componente procurador faz a chamada correta conforme o ambiente escolhido.

4.7.6 Criação dinâmica de objeto remoto

O modelo de programação desta linguagem visual não oferece recursos para a especificação do momento em que os objetos devam ser criados, o que não permite à ferramenta descobrir por si só o momento de instanciação do objeto remoto. O único caso em que isto é possível é no início do programa, onde alguns objetos são criados automaticamente. Fora disto, o usuário deve definir explicitamente quando será executada essa tarefa.

A implementação deste relacionamento é bastante semelhante à anterior. O componente de criação dinâmica possui um método que cria o objeto remoto e a geração de código para o nodo básico acrescenta um método para facilitar essa criação. O arco também é do tipo normal e o mapeamento de propriedades ocorre entre os pares de propriedades (`name`, `remoteName`) e (`host`, `remoteHost`).

A FIGURA 4.21 exibe o método utilizado pelo usuário durante a programação para criar os objetos remotos. A sua finalidade é a mesma do método da seção anterior, ou seja, a de facilitar o uso do componente pois, na verdade, o método do componente de criação dinâmica poderia ser chamado diretamente.

```
InterfaceRemota cr_InterfaceRemota; // componente de criação dinâmica

cr_InterfaceRemota = (InterfaceRemota) cr_createRemoteObject();
```

FIGURA 4.21 - Método que cria o objeto remoto.

Na implementação do componente de criação dinâmica Voyager, foi utilizado o método `Factory.create`, que realiza a criação simples de objetos remotos, sem nenhum tipo de escalonamento ou balanceamento de carga. Neste processo é passado como parâmetro o nome da máquina, fazendo com que a responsabilidade pela distribuição dos objetos se transfira para o programador. Futuramente, com a adição de novas propriedades à ferramenta e da implantação de novos ambientes de execução, escalonamento e balanceamento podem estar presentes.

Um componente de importância fundamental na implementação deste relacionamento é o ambiente de distribuição. Conforme o ambiente utilizado, a criação remota pode ser diferente. Em Voyager, isto é bem simples de se fazer, pois já está disponível uma classe com um método estático que oferece a possibilidade de se criar objetos remotos. Em RMI, entretanto, essa facilidade não existe, exigindo que se utilize um ambiente de execução adicional como o próprio Voyager caso se deseje criar objetos remotos.

4.7.7 Criação de grafo

Quando é criado um relacionamento de criação de grafo, de forma semelhante ao caso anterior da criação de objeto remoto, é inserido no código do nodo básico que irá criar o grafo um método que automatiza essa tarefa. O único trabalho do usuário é chamar esse método no ponto do programa em que isto for necessário.

```
public void createGraph_Grafo() {
    graph_Grafo = new ClasseGrafo();
    String newGraphPath = getGraphPath();
    if (!newGraphPath.equals("")) {
        newGraphPath += ".";
    }
    newGraphPath += "Grafo";
    graph_Grafo.setPath(newGraphPath);
    graph_Grafo.create(newGraphPath);
}
```

FIGURA 4.22 - Método que cria o grafo.

Como se observa na FIGURA 4.22, este método instancia um objeto da classe do grafo chamado, define o caminho do grafo e chama o método `create`, onde estão colocadas as rotinas específicas do ambiente de execução escolhido. Detalhes mais precisos de como é feito o gerenciamento de grafos nas aplicações criadas por esta ferramenta estão na seção 4.8.

4.7.8 Interface de grafo

Quando um grafo é inserido através de um nodo de chamada a grafo, os objetos conectados ao nodo de interface de grafo são exibidos na interface do grafo. Não há nenhuma característica especial desse relacionamento. A sua simples identificação é a informação necessária para que esse relacionamento seja implementado.

4.7.9 Documentação

A implementação do arco de documentação não possui características especiais, tendo apenas a sua função visual, ou seja, a de indicar a qual objeto corresponde determinado nodo de documentação.

4.8 Grafos e subgrafos

Como já foi apresentado, as aplicações criadas com esta ferramenta são estruturadas em grafos. Já que a ferramenta não está amarrada a nenhum ambiente de execução especial de execução e é possível também que o usuário crie os seus próprios tipos de grafos, com geração de código para ambientes de comunicação e execução diversos, a implementação dos grafos não é feita pela ferramenta. A ferramenta apenas define a estrutura e o padrão que a implementação do grafo deve seguir. Para isso foi criada a interface `XGraph`, que define o comportamento que um tipo de grafo deve ter para que a ferramenta o considere como tal.

4.8.1 Objeto XGraph

Para criar os objetos pertencentes aos grafos que formam uma aplicação, a implementação gera uma classe que se responsabiliza pelo seu gerenciamento, executando as seguintes tarefas:

- instanciação remota dos objetos;

- atualização dos parâmetros dos objetos: nome da máquina em que o objeto foi colocado, nome do servidor de nomes e caminho do grafo;
- execução dos nodos básicos, chamando o método `startup` em cada um deles;
- configuração de parâmetros específicos daquele grafo.

Todas essas tarefas são próprias de um determinado ambiente de execução e por isso devem ser implementadas num objeto separado. Caso um outro ambiente deva ser suportado, um objeto que implemente a interface `XGraph` deve ser criado. Estando esta parte da implementação fora da ferramenta, novos ambientes podem ser suportados sem que a ferramenta precise ser alterada. Assim, pesquisadores que construam um novo ambiente de execução distribuída já possuem à sua disposição uma ferramenta de desenvolvimento, caso implementem os objetos específicos de seu sistema. Os métodos da interface `XGraph`, que são invocados pela ferramenta, estão na TABELA 4.22.

TABELA 4.22 - Métodos da interface XGraph.

método	descrição
`generateJavaSource`	gera o código para o grafo
`showConfigDialog`	mostra a janela de configuração dos parâmetros específicos para o grafo

Depois de prontos, os objetos do tipo grafo são disponibilizados para uso na ferramenta. Ao iniciar a edição de um grafo, o usuário seleciona o grafo registrado na ferramenta de acordo com o tipo de aplicação que desenvolverá. Neste momento, um objeto da classe do grafo escolhido é instanciado.

Foram implementados três tipos de grafos (TABELA 4.23). Os tipos de objetos do grafo estão intimamente relacionados com o tipo do grafo. Por exemplo, num grafo do tipo `XVoyagerGraph`, os nodos básicos são do tipo `XVoyagerObject`. Dessa forma, não é possível fazer uma migração automática de um tipo de grafo para outro, ou seja, transformar de forma imediata um grafo desenvolvido para Voyager em um grafo RMI. Toda a estrutura, com objetos e código fonte estão fortemente acopladas.

TABELA 4.23 - Grafos implementados.

grafo	descrição
`XDistributedObjectGraph`	implementa um grafo para instanciação local de objetos
`XVoyagerGraph`	implementa um grafo Voyager
`XRMIGraph`	implementa um grafo RMI

4.8.2 Interface de grafo

A interface de grafo não é percebida diretamente na implementação do programa, isto é, não há código fonte gerado para este objeto visual. Ele é apenas um recurso visual utilizado no editor para indicar se o objeto faz ou não parte da interface do grafo. Quando uma chamada a grafo é inserida em outro grafo, a ferramenta busca as interfaces de grafo e exibe os objetos conectados a elas. Assim, somente estes objetos serão exibidos para interação com os elementos do grafo do qual ele faz parte.

4.8.3 Chamada a grafo

A implementação deste relacionamento é relativamente complexa. Existem várias questões que devem ser estudadas: quando criar um grafo, quando destruí-lo e como realizar a interface com os outros elementos do programa. Além disso, a flexibilidade oferecida com a possibilidade existente de se fazer chamadas recursivas a subgrafos é outro fator que complica a implementação. A vantagem que surge dessa dificuldade é a maior facilidade e flexibilidade para se construir aplicações nesta ferramenta.

Como já foi mostrado na seção 4.6.2.2, existe a interface `XGraphAware` que permite a um objeto fazer parte do contexto de um grafo. Os nodos básicos, componentes de referência remota e de criação dinâmica implementam essa interface, permitindo assim que as referências entre subgrafos funcionem perfeitamente.

O grafo é criado através de um método inserido no código no momento da criação de um relacionamento de interface de grafo (vide seção 4.7.7). Os objetos que implementam a interface `XGraphAware` são identificados por um nome e por um caminho de grafo. Quando um grafo é criado, seus nomes são colocados no servidor de nomes juntamente com o caminho do grafo. Assim, objetos diferentes de mesma classe de grafo são identificados de maneira única dentro da aplicação.

A FIGURA 4.23 ilustra um exemplo dessa situação. Neste exemplo, existem duas instâncias do mesmo tipo de grafo, criados pelo nodo básico `client`. Cada grafo é composto por um nodo básico Voyager e uma interface Voyager. O nodo básico, dentro do seu contexto, é referenciado pelo nome `server`. No contexto da chamada a grafo, ele recebe como prefixo o identificador do seu grafo, para diferenciá-lo do `server` pertencente à outra chamada a grafo. Assim, Os componentes procuradores `XVoyagerProxy1` e `XVoyagerProxy2` referenciam os objetos remotos pelos nomes `G1.server` e `G2.server`, respectivamente.

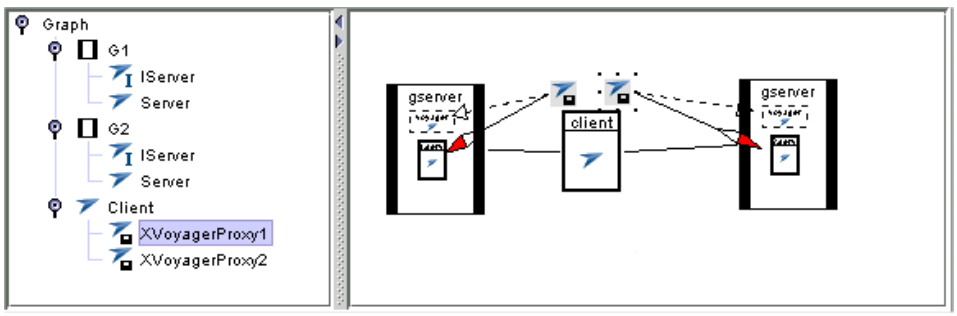


FIGURA 4.23 - Exemplo com chamadas a dois grafos da mesma classe.

A maior parte do processamento dos nomes do grafo é feita logo durante a edição do grafo. Quando um relacionamento é criado entre um objeto do grafo e um outro do subgrafo, na origem o nome do objeto destino recebe como prefixo o nome do grafo.

Existem algumas restrições quanto ao uso de chamadas a grafo. Os únicos objetos que podem ser interfaces de um grafo são o nodo básico, o nodo terminal e o nodo de interface. Além disso, o acesso pode se dar somente de fora para dentro do grafo. Na prática, isto significa dizer que não é possível para objetos de dentro do grafo fazer chamadas remotas a outros objetos fora dele, mas somente a objetos dentro do próprio grafo ou de subgrafos. Essas restrições existem por questões de implementação e facilidade de uso, pois a relação custo/benefício entre o esforço de implementação e os recursos que surgiriam desse esforço não é muito vantajosa. A eliminação dessas restrições pode ser objeto de melhoramentos futuros da ferramenta.

4.9 Geração de código

A geração de código feita aqui é diferente daquela feita nas ferramentas mais conhecidas de programação visual para ambientes paralelos e distribuídos. Ela é mais parecida com aquela implementada nas ferramentas de programação visual para Java, como o JBuilder, onde é possível alterar ao mesmo tempo tanto a parte visual como a parte textual dos programas. As alterações na parte visual são tratadas pela ferramenta, que gera o código textual correspondente ao grafo e o programador altera o código textual para colocar a lógica do seu programa (a forma como as duas representações são mantidas coerentes será demonstrada na seção 4.9.2).

TABELA 4.24 - Geração de código nas ferramentas visuais paralelas e distribuídas e em Java.

característica	programação paralela e distribuída	Java
tempo de geração	geração do código ao final da programação	geração de código durante a construção da aplicação
implementação	geração de código específico para a ferramenta	geração de código padrão Java
tipo de aplicação	paralela e distribuída	aplicação genérica
flexibilidade	o usuário não pode alterar o código gerado	o usuário pode alterar o código gerado
integração com outras ferramentas	o código gerado só pode ser mantido pela ferramenta original	o código pode ser processado por outras ferramentas, pois o código gerado é padrão Java

A TABELA 4.24 compara as características da implementação da geração de código nas ferramentas de programação visual paralela e distribuída tradicionais com a das ferramentas que geram código para Java. A maioria das diferenças entre os dois tipos de ferramentas vem do fato de que no primeiro tipo é possível se trabalhar apenas sobre a descrição visual do programa, sendo todo o resto controlado pela ferramenta. Nas ferramentas do segundo tipo, o programador pode trabalhar ao mesmo tempo sobre a representação visual e sobre a textual, deixando para a ferramenta a responsabilidade pela manutenção da coerência entre essas duas representações.

4.9.1 Implementação

Como está no capítulo 3, o modelo da ferramenta prevê que a geração do código dos nodos básicos e dos nodos de interface seja implementada nos próprios nodos e não dentro da ferramenta, a qual apenas comanda a geração do código. Sendo assim, o fluxo da geração de código para um objeto passa por várias entidades, conforme a seguinte seqüência de passos:

- o usuário insere um nodo básico ou nodo de interface no grafo e a ferramenta cria uma nova janela de edição de texto para o código do nodo inserido;
- o nodo básico ou de interface inserido contém por definição um gerador de código. Logo após a inserção, a ferramenta requisita a este gerador a criação do código inicial do objeto a ser colocado na janela de edição de texto. No caso do nodo básico, por exemplo, é inserido um esqueleto com a declaração da classe;
- quando o usuário executa alguma ação no editor de grafos que cause alteração no código, a ferramenta chama o gerador de código do nodo básico com a ação

e seus parâmetros. A ferramenta passa ao gerador de código o texto atual do código juntamente com a ação, recebendo o novo texto com as alterações;

- se o usuário remove o nodo básico ou de interface, a janela de edição textual também é removida;
- a qualquer momento o usuário pode também editar o código fonte diretamente através do editor textual. Entretanto, há algumas restrições, como não poder apagar nem alterar o código gerado (mais detalhes sobre isso virão a seguir).

Tudo isso exige portanto que haja uma interface conhecida entre a ferramenta e o gerador de código para que eles se entendam. Para isso foi definida a interface `XCodeGenerator`, que abrange o conjunto de ações que podem ser executadas durante a edição do programa visual e que devem ser implementadas pelo gerador de código. A TABELA 4.25 mostra os métodos da interface `XCodeGenerator`.

TABELA 4.25 - Métodos da interface XCodeGenerator.

métodos	**momento em que são chamados**
`on [Create/NameChange]`	[inserção/alteração do nome] de um nodo básico no grafo
`onPropertyChange`	alteração do valor de uma propriedade
`onEventNameChange`	alteração do método de tratamento de um evento
`onComponent [Insert/Remove/NameChange]`	[inserção/remoção/alteração do nome] de um componente
`onMethod [Insert/Remove/NameChange]`	[inserção/remoção/alteração do nome] de um método de um nodo básico
`onInterface [Insert/Remove/NameChange]`	[inserção/remoção/alteração do nome] de um nodo de interface
`onInterfaceMethod [Insert/Remove/NameChange]`	[inserção/remoção/alteração do nome] de um método de interface
`onProxyInterface [Insert/Remove/NameChange]`	conexão de um nodo de interface a um nodo procurador, desconexão ou mudança do nome desse nodo de interface
`onDynamicInterface [Insert/Remove/NameChange]`	conexão de um nodo de interface a um nodo de criação dinâmica, desconexão ou mudança do nome desse nodo de interface
`onGraphCreation [Insert/Remove/NameChange/ ClassNameChange]`	[inserção/remoção] de um arco de criação de grafo num nodo básico. [mudança do nome do nodo de grafo/mudança da classe do grafo] conectado ao nodo básico
`getInterfaceMethod Declaration`	retorna a declaração de um método de interface que deve ser inserido no código do nodo básico

Estes são os métodos que um gerador de código deve implementar. Todos esses métodos recebem como entrada o texto atual e retornam o texto modificado. Como se vê, não é dito nada a respeito de como deva ser a implementação interna de cada método. Isto será visto na próxima seção, onde será tratado o problema da consistência entre a representação visual e textual do programa.

4.9.2 Consistência entre código textual e visual

Pelo fato da ferramenta combinar representações visuais e textuais de uma mesma especificação, uma questão importante que deve ser levada em conta é a consistência entre tais representações. A ferramenta deve garantir ao programador que as alterações que ele fizer em uma das representações sejam refletidas na outra de forma consistente. No caso desta ferramenta, onde são utilizadas representações textuais e visuais da mesma especificação, o desafio é fazer com que o grafo da aplicação seja consistente com o código textual gerado que o implementa. Este problema é relativamente complexo e se torna ainda mais complicado quando se permite acesso livre do usuário à edição simultânea das duas representações.

Foram pesquisados algoritmos para tratamento dessa questão, mas não foi encontrada documentação sobre o assunto. Sendo assim, foi desenvolvido um algoritmo simples para manter a coerência entre as duas representações. No algoritmo, o gerador de código estrutura o código fonte do nodo básico ou de interface em uma lista contendo as informações sobre seus componentes, propriedades e eventos definidos a partir da edição gráfica da aplicação. No momento de se criar o código fonte, o gerador percorre a lista e monta o código com os trechos da declaração do componente, da atualização dos valores das propriedades, dos métodos de registros de eventos e de outras características específicas do tipo de componente. Quando um novo componente é adicionado ou removido do objeto, o gerador de código é chamado para processar a alteração, atualizando a sua estrutura de dados e finalmente o código gerado.

A estrutura do algoritmo descrita até agora seria suficiente para um gerador de código de uma ferramenta onde não fosse possível se mexer sobre o código fonte gerado, pois a base para a geração parte somente de uma representação da aplicação (a representação visual). Entretanto, é permitido ao programador trabalhar sobre o código fonte gerado diretamente. Por isso, ao modelo inicial de geração de código são acrescentadas características complementares.

O primeiro passo é fazer a diferença entre o código gerado e aquele inserido pelo programador. Para isto, o gerador de código acrescenta ao final de cada linha que ele gera uma seqüência especial de caracteres que não pode estar presente no código do usuário (nesta implementação foi escolhida a seqüência “//#”). Se esta restrição for violada, o gerador de código irá tratar o código do usuário como sendo código gerado e provavelmente surgirão erros no processamento das alterações do código fonte.

Feita essa diferenciação, o próximo passo é fazer com que as linhas de código inseridas pelo programador antes da chamada do gerador de código sejam preservadas após a nova geração. Neste ponto, o gerador terá duas representações do código fonte: um novo código fonte gerado a partir da estrutura do código na lista explicada acima que não contém o código do usuário e o próprio código fonte textual original, com o código gerado e o código inserido pelo programador antes da alteração. A tarefa a ser feita a partir deste momento é combinar essas duas representações para produzir o código fonte final.

Para facilitar esse trabalho, os códigos fontes a serem combinados são analisados linha a linha e as operações possíveis sobre o código fonte foram classificadas em três tipos: inserção, remoção e alteração de uma linha. Essas operações são desencadeadas durante a edição visual dos grafos da aplicação e são tratadas de maneira diferente pelo gerador de código para produzir o código textual final:

- **inserção de linhas de código**: quando o usuário insere um novo componente num nodo básico, o gerador de código atualiza a lista da estrutura do código fonte. Para construir o código fonte final, ele compara a linha que deve ser escrita com a linha do código fonte original que está sendo analisada no momento. Se a linha analisada está também no código original, então a escreve no código final e avança para a próxima linha nos dois códigos. Se a linha atual no código original foi inserida pelo usuário, então escreve essa linha no código final e avança para a próxima linha no código original. Se aparecer uma linha gerada no código original, então compara novamente com a linha gerada do novo código. Se as linhas forem iguais, chegou-se à situação já descrita acima. Caso contrário, então significa que a linha do código estruturado é nova em relação ao código original e deve ser escrita no código final. Essas situações ocorrem quando uma operação na representação visual causa a inclusão de código, como por exemplo a inserção de um novo componente, a alteração de uma propriedade que tinha o valor ***default*** ou a inclusão de um método de tratamento de evento;
- **remoção de linhas de código**: se a linha analisada no momento no código original for uma linha inserida pelo usuário, então a escreve no código final. Se for gerada pela ferramenta e ela for igual à linha atual no novo código gerado, então a escreve no código final. Entretanto, se as linhas forem diferentes, significa que a linha do código original foi removida. Assim, a linha do código original é ignorada até que apareça alguma que seja igual ao do código estruturado ou todo o código original já tenha sido percorrido;
- **alteração do conteúdo de uma linha de código**: o procedimento tomado neste caso é semelhante ao caso da inserção. Na inserção, quando uma linha gerada do novo código for diferente do código antigo, então as duas linhas são inseridas no código. Na alteração, somente a nova linha é repassada para o código final, pois se as linhas são diferentes então significa que o código foi alterado (considerando-se que a análise das linhas dos arquivos esteja sincronizada).

Este algoritmo é simples e por este motivo tem algumas restrições:

- não é possível alterar linhas do código gerado;
- as linhas inseridas pelo usuário não podem terminar com “//#”;
- a estrutura do código não pode ser perdida, ou seja, a representação visual deve ser mantida (a representação visual não pode ser reconstruída apenas a partir do código fonte).

Essas restrições não impedem o usuário de desenvolver qualquer tipo de programa. Por isso, considerou-se adequada a solução escolhida para uma versão inicial da ferramenta. Numa situação ideal, o usuário poderia executar as alterações que entendesse necessárias em quaisquer das representações sem se preocupar com a consistência e a ferramenta trataria este problema. Raras ferramentas realizam esta tarefa de forma totalmente transparente ao usuário. A ferramenta JBuilder, por exemplo, atinge esse objetivo através de uma análise completa do código fonte do usuário, praticamente executando o código em tempo de desenvolvimento. Essa solução, além de ser muito pesada, exige uma implementação bastante complexa.

4.9.3 Geração de código para o grafo

A geração de código para o grafo é tarefa do objeto que implementa a interface `XGraph`, isto é, não é implementada dentro da ferramenta. Sendo assim, não há um padrão para este procedimento, exceto que o grafo deve implementar a interface `XRunGraph`, que contém os métodos `setGraph`, `getGraph` e `create`. A FIGURA 4.24 mostra o código gerado para o grafo da FIGURA 4.26.

```
import vpj.*;
import vpj.base.*;
import vpj.graph.*;

public class chat extends XDefaultRunGraph implements XRunGraph {
  public void create(String path) {
    String host;
    String address;
    String prefix;

    super.create(path);
    if (path.equals("")) {
      prefix = "";
    } else {
      prefix = path + ".";
    }
    XBaseNode obj0 = (XBaseNode) new Client();
    obj0.setAllProperties(null, path, "localhost", "localhost:8000");
    obj0.startup();
    XBaseNode obj1 = (XBaseNode) new Server();
    obj1.setAllProperties(null, path, "localhost", "localhost:8000");
    obj1.startup();
  }

  public static void main(String args[]) {
    chat graph = new chat();
    graph.create("");
  }

}
```

FIGURA 4.24 - Código gerado para o grafo da FIGURA 4.26.

Como auxílio, existe a classe `XDefaultRunGraph` que contém uma codificação padrão dos métodos da interface `XRunGraph`. Assim, no exemplo, somente o método `create` foi redefinido para a criação dos nodos básicos do grafo (o nodo de serviço do grafo é criado dinamicamente pelo nodo básico `server`). Este grafo é do tipo normal, onde os objetos são simplesmente instanciados localmente.

O método `setAllProperties` faz parte da estrutura dos nodos básicos e é usado para definir as principais propriedades dos nodos básicos no momento da criação do grafo. Essas propriedades são o nodo pai (no caso, `null`, pois os objetos estão no nível mais alto de instanciação), o caminho do grafo (passado como parâmetro na chamada do método `main` e igual a “”, pois estão no nível mais alto do grafo), o nome da máquina em que se encontram e o endereço do servidor de nomes (não utilizado neste exemplo).

A geração de código para o grafo é bem mais simples do que para os nodos básicos, pois o usuário não interage com o código gerado para um grafo. É um processo de coleta de dados e produção do código, sem interferência do usuário no meio do processo. Para permitir que ele seja chamado diretamente da linha de comando e não apenas de outros objetos, o método estático `main` pode ser implementado.

4.9.4 Geração de código para o projeto

A geração de código para o projeto também é simples. Já que a cada projeto está associado um grafo que é o seu ponto de entrada, basta ao projeto criar um objeto desse tipo, como mostra a FIGURA 4.25.

```
public class apv {
    public apv() {
        apv graph = new apv();
        graph.create("");
    }

    public static void main(String args[]) {
        apv project = new apv();
    }
}
```

FIGURA 4.25 - Código gerado para um projeto.

O projeto representa toda a aplicação. Portanto, para executá-la, basta passar ao interpretador Java o nome do projeto, cuja classe, como mostra a figura acima, possui o método estático `main`.

4.10 Outros detalhes de implementação

Nesta seção serão apresentados alguns detalhes menos importantes da implementação, mas que ainda assim valem a pena serem mencionados.

4.10.1 Configuração visual

O emprego de uma linguagem visual no desenvolvimento de programas não serve apenas para especificar a concorrência com as suas características de bidimensionalidade, mas também para apresentar as informações de uma maneira mais rica e agradável, com a possibilidade de utilização de cores, gráficos e outros recursos visuais que ajudem o programador a visualizar melhor sua aplicação. Assim, todos os componentes podem ser configurados para que apresentem ao usuário as informações da melhor forma que lhe convier, contendo configurações que lidam com as cores e os textos que compõem os objetos. Além disso, a linguagem visual proposta contém objetos especiais cujo único objetivo é o enriquecimento visual do grafo.

Quanto à representatividade dos objetos, procurou-se na medida do possível implementar a visualização dos nodos e arcos de acordo com a linguagem de modelagem UML [FOW97]. Programadores acostumados a essa linguagem podem compreender com mais facilidade os grafos dessa ferramenta.

4.10.2 Terminação das aplicações

Não há um comando ou método específico para terminar toda a aplicação e seus objetos automaticamente. Portanto, o programador deve se preocupar diretamente com a terminação de objetos que não se encerram automaticamente. Um exemplo é a porta de comunicação. Quando criada, ela gera *threads* que devem ser fechadas ao final da aplicação. Nesse caso, o programador deve chamar o método `close` da porta, assim como deve chamar o método `shutdown` quando se tratar de nodos básicos.

4.10.3 Desenvolvimento de novos componentes

O desenvolvimento de novos componentes exige apenas que se sigam os padrões da arquitetura de componentes JavaBeans e se implementem as interfaces correspondentes aos tipos de objetos desejados, como descrito em seções anteriores.

4.11 Exemplos de aplicações

Para validar a ferramenta, foram implementados alguns exemplos simples a fim de se demonstrar a flexibilidade da ferramenta e o que é possível de se fazer com ela. Entretanto, a fim de evitar excesso de detalhes, serão apresentados apenas os grafos das aplicações. A aplicação completa pode ser obtida junto à distribuição da ferramenta.

Exemplos já foram dados de como se realiza a comunicação por *sockets* e por meio de invocação remota de métodos (seções 4.7.1 e 4.7.5, respectivamente). A partir de agora serão dados exemplos de aplicações que utilizam estes tipos de comunicação, da forma como estão providas nesta ferramenta.

4.11.1 Bate-papo com *sockets*

Um exemplo tradicional de programação distribuída cliente-servidor implementada com *sockets* é o programa de bate-papo. Nesta aplicação, um servidor fica disponível permanentemente e atende a requisições de diferentes clientes.

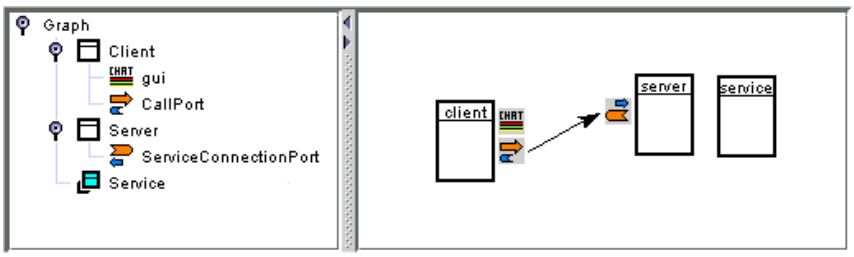


FIGURA 4.26 - Bate-papo implementado com sockets.

Neste exemplo se verificam os recursos disponíveis ao desenvolvimento de aplicações cliente-servidor com *sockets*. São utilizados os componentes porta de chamada de serviço no nodo básico `client` e a porta de conexão de serviço no objeto `server`. O nodo básico de serviço `service` é criado dinamicamente pela porta de conexão de serviço e contém o código que trata a requisição.

4.11.2 Bate-papo com chamadas remotas de método

Para mostrar as diferenças entre trocas de mensagens e chamadas de método neste modelo, segue o mesmo exemplo da aplicação de bate-papo, mas agora implementada com chamadas de métodos remotos.

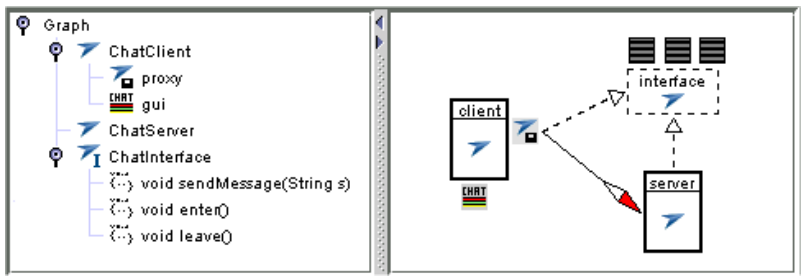


FIGURA 4.27 - Bate-papo implementado com chamadas de métodos remotos.

Agora, o cliente faz as chamadas diretamente no servidor. Em termos de visualização, a representação neste caso fica até mais complicada do que com *sockets*, mas certamente a implementação do usuário fica muito mais simples, pois não terá que lidar com trocas de mensagens no ambiente distribuído.

Quanto à implementação, ao contrário do exemplo anterior, o código que trata a requisição é colocado dentro do objeto `server`. No exemplo com *sockets*, o nodo básico `server` é responsável apenas pela criação de um novo serviço, deixando a implementação para o objeto `service`.

4.11.3 Jantar dos filósofos

O problema do jantar dos filósofos é um problema clássico de programação concorrente e por isso mesmo é encontrado na maioria dos livros de sistemas operacionais. Em resumo, consiste de cinco filósofos à mesa de jantar onde estão disponíveis cinco garfos. Para levar a comida à boca, um filósofo deve usar o garfo que está à sua direita e à sua esquerda. O objetivo do problema é se analisar como se comportam a requisição e a liberação de recursos compartilhados em função de determinados esquemas e primitivas de sincronização utilizados na sua implementação (a descrição completa do problema pode ser encontrada em [TAN97]).

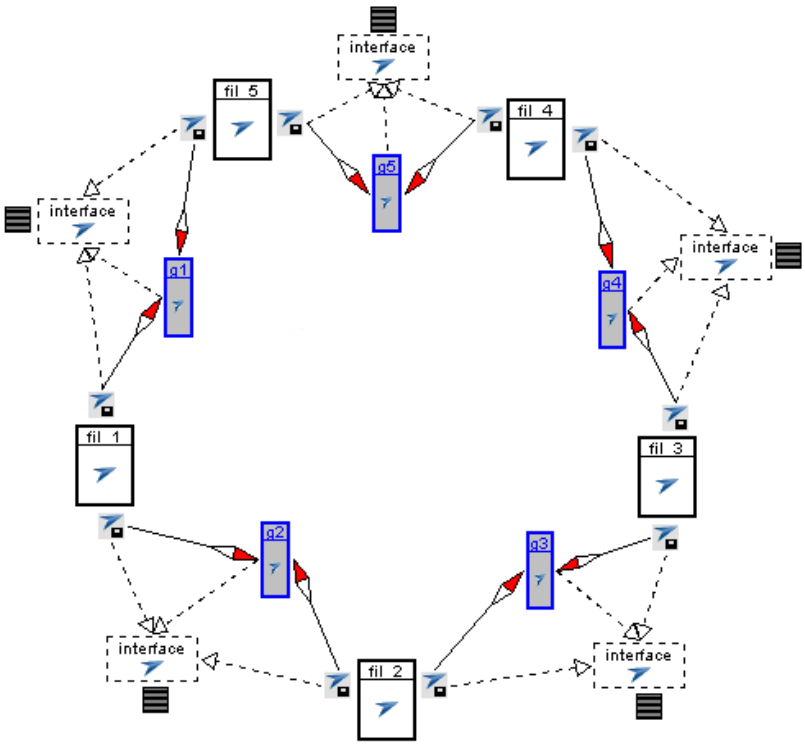


FIGURA 4.28 - Problema dos filósofos.

No exemplo, este problema foi implementado com chamadas de métodos remotos. Os garfos são os objetos servidores de métodos remotos. A sincronização ocorre no momento em que um método que já está sendo executado não pode ser chamado por outro objeto simultaneamente. Esse exemplo mostra também como a configuração visual pode ser importante. Como se vê na FIGURA 4.28, a forma como são construídos graficamente os objetos, com diferentes cores e tamanhos, auxilia bastante na representação e compreensão do programa (os objetos menores mais ao centro - g1 a g5 - são os garfos e os mais externos - fil 1 a fil 5 - são os filósofos).

4.11.4 Quicksort

O algoritmo de classificação de dados Quicksort é bastante conhecido. Sua presença neste trabalho se deve à sua implementação recursiva, que permite demonstrar o uso prático da chamada recursiva de grafos.

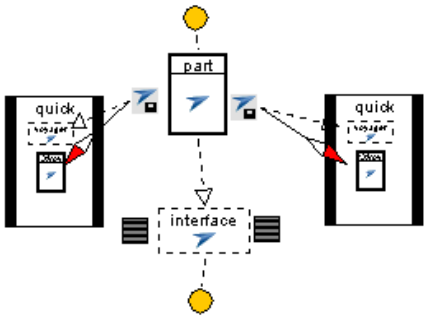


FIGURA 4.29 - Quicksort com chamada recursiva de subgrafos.

A FIGURA 4.29 mostra o grafo do algoritmo *quicksort* desenvolvido nesta ferramenta. Um objeto do grafo realiza uma chamada remota a outro objeto do subgrafo, que é da mesma classe. Antes disso, entretanto, ele cria o grafo que irá tratar a requisição. A parte que faz o particionamento e distribuição dos valores está implementada no nodo básico *part*, que é invocado na próxima chamada do grafo, mas com outra instância. Assim, a classificação das partes dos vetores que ficam antes e depois do elemento particionador é feita pelos subgrafos da esquerda e da direita, respectivamente.

4.12 Resumo do capítulo

Este capítulo mostrou como foi feita a implementação do modelo apresentado no capítulo anterior. Esses dois capítulos, por tratarem do projeto e implementação da ferramenta, são os mais importantes de todo este trabalho.

O capítulo se iniciou pela descrição geral da implementação e pela explanação dos motivos pelos quais foram escolhidas as ferramentas usadas para tal. Depois disso, mostrou a implementação e o funcionamento da interface gráfica da ferramenta, principalmente do editor de grafos.

A seguir, vieram os detalhes de como foram construídos os objetos da linguagem visual e os seus relacionamentos, ou seja, os arcos. Pela complexidade e importância dos temas, foi dado tratamento especial a dois pontos da implementação: grafos e geração de código.

Para finalizar, foram apresentados grafos de algumas aplicações para se dar uma visão mais prática de como uma aplicação é estruturada nessa ferramenta.

5 Trabalhos relacionados

Como já foi visto anteriormente, o ambiente de programação apresentado aqui pode ser comparado com vários outros tipos de ambientes também usados na programação distribuída. O objetivo da comparação apresentada neste capítulo é realçar a contribuição oferecida pelo trabalho e mostrar os motivos pelos quais tal contribuição é positiva. Além disso, logicamente, deve-se citar os seus pontos fracos e desvantagens em relação aos outros projetos a fim de orientar o usuário na escolha de um ambiente que satisfaça às suas necessidades.

5.1 Estrutura da comparação

As áreas em que este ambiente se situa (programação visual, programação distribuída, programação em Java) permitem que ele seja comparado com três tipos de ferramentas de programação:

- **ferramentas de programação textual**: linguagens de programação textual (linguagens concorrentes) e bibliotecas de classes para linguagens textuais seqüenciais. Exemplos: CC++, PVM, MPI;
- **ferramentas de programação visual paralela e distribuída**: ferramentas onde se enquadram HeNCE, CODE, VisualProg e VPE, entre outras;
- **ferramentas de programação visual em Java**: ferramentas usadas principalmente em aplicações *destktop*, como JBuilder, IBM VisualAge, Symantec Visual Café e Microsoft Visual J++.

Cada comparação enfatizará pontos escolhidos conforme o relacionamento entre as ferramentas comparadas. Por exemplo, no primeiro caso, será dada ênfase ao valor da linguagem visual na programação distribuída, pois é esta a principal diferença entre este trabalho e as ferramentas de programação textual.

A estruturação da comparação em três partes favorece a compreensão das razões pelas quais as decisões de projeto foram tomadas neste trabalho. Inicialmente, entretanto, será feita uma análise das vantagens e desvantagens dessa ferramenta de maneira genérica. Tendo em vista que determinadas características podem ser comparadas uniformemente com todas as outras ferramentas, uma análise deste tipo é válida.

5.2 Comparação geral

A ferramenta apresenta muitas funcionalidades que são encontradas em vários tipos de ambientes de programação. Procurou-se tomar de cada um as características que fossem benéficas ao desenvolvimento de aplicações distribuídas e integrá-las a fim de oferecer um ambiente único mais adequado a esses tipos de aplicações. Analisando-se o contexto das aplicações distribuídas e dos ambientes de programação em geral, pode-se tirar algumas conclusões sobre vantagens e desvantagens globais em relação aos outros ambientes de desenvolvimento.

5.2.1 Vantagens

Determinados aspectos podem ser apontados em relação a todos os outros trabalhos como vantagens. Embora alguns dos ambientes apresentem essas

características, elas serão colocadas neste espaço com o objetivo de enfatizar a importância delas no ambiente desenvolvido. As mais relevantes são:

- **programação orientada a objeto**: oferece aos programas uma série de benefícios (encapsulamento, herança, polimorfismo), facilitando o desenvolvimento de aplicações mais complexas, em particular as aplicações distribuídas;
- **modelo de componentes de *software***: permite que haja reutilização de código e facilita a extensão do sistema. O usuário que deseja acrescentar uma característica nova à ferramenta deve apenas implementar um novo componente. A padronização das interfaces da arquitetura de componentes facilita o trabalho de integração deste componente à ferramenta;
- **portabilidade**: tanto a ferramenta como o código com que o usuário trabalha são portáveis entre diferentes plataformas, graças ao ambiente Java;
- **geração automática de código**: parte do código é responsabilidade da ferramenta, o que diminui a chance do aparecimento de erros.

As características acima são resultado da busca de se desenvolver um ambiente de programação que ofereça atenção especial ao desenvolvimento de *software*. Procura-se assim evitar o que acontece em muitas ferramentas de programação paralela e distribuída, onde esses aspectos são deixados de lado e a qualidade e produtividade do desenvolvimento são prejudicadas.

5.2.2 Desvantagens

É praticamente impossível se construir uma ferramenta de programação de caráter abrangente, aplicável em muitas áreas, e que consiga ainda assim suplantar as ferramentas especializadas nos aspectos mais pontuais. A criação de novas ferramentas que ataquem pontos mais específicos tem justamente o objetivo de oferecer mais recursos aos usuários através da especialização das atividades. Este é um ponto interessante, visto que determinados fabricantes apontam a integração de ferramentas como uma vantagem e outros como uma desvantagem, pois estes acreditam mais na especialização. Certamente cada um possui a sua parcela de razão. O contexto em que serão usadas é que irá determinar o peso de cada aspecto analisado.

Em vista da concorrência com as ferramentas de uso mais específico, assim como há vantagens que podem ser apontadas em relação a todas as outras ferramentas, desvantagens também fazem parte dessa comparação:

- **fraco desempenho de Java**: a execução de Java é interpretada, tornando o desempenho o seu ponto mais fraco. Algumas soluções vêm sendo apresentadas para minimizar este problema, como compiladores *just in time* e a tecnologia de otimização dos programas desenvolvida pela Sun, denominada HotSpot [SUN2000f]. Entretanto, ainda estão muito longe do desempenho alcançado pelos programas compilados para código objeto nativo;
- **curva de aprendizado**: embora a ferramenta tenha sido projetada com o objetivo de facilitar o seu uso, muitos usuários certamente preferirão continuar utilizando as suas antigas ferramentas (o que vêm fazendo com certo sucesso) a ter que investir numa nova opção. Isto se deve a fatores pessoais (preferência por um tipo de ferramenta, experiência anterior) e técnicos (compatibilidade entre ferramentas, possibilidade de uso de código legado).

Assim como acontece nas vantagens, os pontos apresentados acima são resultado da análise geral de ambientes de programação. A seguir, serão feitas comparações mais específicas, apresentando os pontos positivos e negativos em relação às ferramentas de programação textual, ferramentas de programação visual paralela e distribuída e ferramentas de programação visual em Java.

5.3 Ferramentas de programação textual

As ferramentas de programação textual ainda são as mais utilizadas na programação de ambientes paralelos e distribuídos. O grande número de ferramentas desse tipo e a cultura em torno delas contribuem para que se mantenha esse *status* e se imponham restrições à aceitação de novas ferramentas. Esse é o principal motivo pelo qual a introdução de novas ferramentas com diferentes paradigmas de programação ocorre de maneira bastante tímida. Como exemplo, embora seja reconhecidamente trabalhoso, o método de depuração através de impressão de mensagens na tela ainda é um dos métodos de depuração mais usados entre os programadores de aplicações concorrentes. Eles ainda preferem utilizar esta técnica pouco elegante a ter que aprender a utilizar uma nova ferramenta.

5.3.1 Vantagens

Com relação às ferramentas textuais de programação concorrente (linguagens concorrentes, compiladores etc), esta ferramenta apresenta ao programador as seguintes vantagens:

- a principal vantagem e objetivo deste trabalho é a chance de se verificar com maior clareza os relacionamentos entre os diferentes objetos distribuídos da aplicação. Os gráficos possuem uma característica própria (bidimensionalidade) que favorece essa tarefa, (muito complicada em linguagens de programação textual);
- visualização da estrutura da aplicação: facilita o controle do projeto, mostrando os objetos e componentes participantes de forma rápida e clara;
- a linguagem de programação (Java) já é conhecida, o que permite a utilização de pessoal e de programas já existentes. Embora muitas linguagens concorrentes se baseiam em linguagens convencionais conhecidas (C, C++), outras são totalmente novas, dificultando a reutilização de programas, como por exemplo SR e Linda;
- a linguagem é orientada a objeto, o que não acontece em todas as linguagens concorrentes;
- componentes de *software*: é possível se ter componentes em linguagens textuais, mas os componentes para programação distribuída ainda são raros;
- eficiência: com relação ao paralelismo e distribuição implícitos (compiladores especiais), a especificação explícita da concorrência produz programas mais eficientes.

O fato de a ferramenta combinar a visualização com a programação textual faz com que ela tenha a mesma flexibilidade dos programas textuais. Embora a ferramenta controle parte do código, o programador está livre para fazer alterações que desejar diretamente no código, o que dá o poder que muitos usuários precisam para expressar tudo o que precisam e do jeito que desejam.

5.3.2 Desvantagens

O fato de ferramentas novas como esta terem sua aceitação restrita tem as suas causas. As desvantagens em comparação com as ferramentas mais usadas atualmente, as ferramentas textuais de programação paralela e distribuída, são principalmente:

- a maior parte do *software* já existente é textual, facilitando o reaproveitamento de código nas ferramentas textuais. Apesar desta ferramenta combinar a parte visual com a representação textual, partes desenvolvidas por terceiros que expressem concorrência serão encapsuladas e não serão visualizadas graficamente. Além disso, a linguagem Java é relativamente nova, perdendo para outras como C e C++ na quantidade de *software* desenvolvido;
- eficiência: Java não é eficiente e isso certamente é um problema para muitos usuários, principalmente em se tratando de programação paralela;
- ambiente pesado: os ambientes de programação mais complexos são muito mais pesados do que um compilador de linha de comando. Além de exigir uma interface gráfica, as funcionalidades adicionais exigem mais recursos da máquina. Isto é parte do preço que se deve pagar pelas facilidades extras;
- comparando-se com os compiladores que automatizam a distribuição e o paralelismo, nesta ferramenta isto deve ser feito explicitamente pelo programador, o que se constitui numa tarefa a mais para o usuário;
- simplicidade das aplicações: para aplicações pequenas ou cujas necessidades de distribuição e comunicação são simples, o esforço necessário de adaptação a uma nova ferramenta pode não valer a pena.

Esses pontos são suficientes para que muitos usuários desistam de utilizar uma nova ferramenta. Entretanto, outros motivos que serão melhor discutidos na conclusão são ainda mais importantes para tal decisão.

5.4 Ferramentas de programação visual paralela e distribuída

Outro importante objetivo deste trabalho é superar as deficiências dos ambientes atuais de programação paralela e distribuída. Esses ambientes de programação enfatizam a programação paralela (com geração de código para PVM e MPI) e pecam pela falta de suporte mais elaborado a aplicações clássicas de sistemas distribuídos, como sistemas cliente-servidor. Além da deficiência na geração das aplicações, questões de engenharia de *software* como reutilização de código, orientação a objeto e portabilidade são mal tratadas ou completamente ignoradas.

5.4.1 Vantagens

A maior parte da motivação deste trabalho foi buscada nas deficiências das ferramentas visuais de programação paralela e distribuída vigentes. Através da busca das soluções dos problemas dessas ferramentas, chega-se às vantagens que resultam dessa troca:

- programação orientada a objeto: a maioria das ferramentas de programação visual paralela e distribuída utiliza a linguagem C que, ao contrário de Java, não é orientada a objeto;

- objetos distribuídos: a utilização de Voyager como meio de comunicação e ambiente de execução permite a instanciação remota de objetos e invocação remota de métodos nesses objetos, o que não existe nas outras ferramentas;
- programação orientada a eventos: em programação distribuída, determinadas tarefas são difíceis de serem programadas utilizando-se o paradigma convencional. Por exemplo, a menos que se fique constantemente realizando testes, é difícil saber quando algum erro ocorreu na comunicação. Com eventos, o objeto interessado nessa informação não se preocupa com isso até que seja notificado da ocorrência do erro;
- grande possibilidade de extensão da ferramenta: a generalização da ferramenta e a especialização das atividades através dos componentes permitem que novas soluções sejam incorporadas sem que alterações necessitem ser feitas no núcleo da ferramenta. Para isto, basta apenas a implementação e adição dos componentes;
- a utilização de componentes colabora também para o desenvolvimento modular e reutilização de código;
- flexibilidade: a ferramenta gera código textual no qual o usuário pode trabalhar livremente e inserir a sua parte conforme a necessidade. Outras ferramentas são muito restritas e se adaptam somente a um determinado tipo de aplicação. Isto acontece nas ferramentas CODE, HeNCE e VisualProg, por exemplo, onde a comunicação é restrita ao início e ao fim de blocos de computação;
- portabilidade da ferramenta: graças ao ambiente Java, a ferramenta pode executar sem modificações na maioria das plataformas de *hardware* e *software* correntes;
- portabilidade das aplicações: assim como acima, Java permite que as aplicações executem em qualquer ambiente suportado por Java. As outras ferramentas buscam portabilidade através do PVM e MPI que existem em muitas plataformas, mas isso exige pelo menos a recompilação do código fonte das aplicações;
- escalabilidade da linguagem visual: através da utilização de grafos e subgrafos é possível diminuir bastante a complexidade da representação visual dos programas. Nem todas as ferramentas oferecem esse suporte;
- execução de uma aplicação simultaneamente em diferentes ambientes: é possível se ter numa mesma aplicação diferentes tipos de objetos, onde cada um utiliza o seu próprio ambiente de execução. A comunicação entre esses objetos, entretanto, deve seguir um protocolo comum, como por exemplo TCP;
- não há uma sintaxe adicional para anotação dos elementos do programa, como acontece em muitos ambientes de programação visual paralela e distribuída. A exigência do conhecimento de uma sintaxe especial é um empecilho considerável ao uso fácil dessas ferramentas;
- suporte a diferentes tipos de comunicação: a comunicação entre objetos distribuídos se caracteriza pela invocação remota de método, mas nesta ferramenta é possível também realizar comunicação através de *sockets* e invocação local de métodos (memória compartilhada);

- programação cliente-servidor: o gerenciamento de conexões, com criação automática de objetos para tratar as requisições, é uma característica não encontrada nos outros ambientes de programação.

Todas essas vantagens são resultado da tentativa de se suprir as deficiências de boas ferramentas visuais para a programação distribuída. Acredita-se que elas sejam suficientes para que se considere este ambiente mais apropriado do que os outros no campo da programação distribuída.

5.4.2 Desvantagens

Por estarem há mais tempo em uso e desenvolvimento e assim recebido com maior qualidade o retorno do usuário, as ferramentas atuais de programação visual paralela e distribuída possuem algumas funcionalidades que ainda não estão presentes aqui. As principais deficiências destes trabalhos em relação àquelas ferramentas são:

- falta de visualização: praticamente todas as ferramentas de programação paralela apresentam um sistema de monitoração e visualização da execução. Esta quase unanimidade deve-se à importância que a visualização tem para a programação paralela, onde a visualização é utilizada na sintonia do desempenho e também como auxílio à depuração;
- falta de depuração: embora mais rara do que a visualização, a depuração paralela pode ser encontrada em alguns ambientes;
- no caso específico da programação paralela, atualmente as ferramentas baseadas em Java são praticamente descartadas em função da ineficiência;
- ainda que se desconsidere o fato da baixa eficiência, a ferramenta oferece pouco suporte a aplicações paralelas. Não apresenta, por exemplo, replicação de objetos com vistas à exploração do paralelismo de dados, o que pode ser encontrado em outros ambientes;
- determinadas ferramentas possuem características mais específicas, devido ao tipo especial de aplicação a que foram destinadas: tolerância a falhas, replicação de objetos, balanceamento de carga, comunicação de grupo.

As desvantagens acima podem ser consideradas como sugestões de trabalhos futuros a esta dissertação, assim como outras atividades que estão destacadas na conclusão.

5.5 Ferramentas de programação visual em Java

Neste espaço será feita uma comparação entre as ferramentas de programação visual existentes para Java e o trabalho proposto. Dirigidas principalmente para aplicações *desktop*, essas ferramentas são ambientes de programação dotados de muitos recursos e que por esses e outros motivos atingiram grande sucesso comercial. Estão há um certo tempo no mercado e a concorrência entre elas garante o lançamento contínuo de novas versões com cada vez mais funcionalidades.

5.5.1 Vantagens

O motivo que leva a esta comparação é mostrar os problemas não resolvidos por essas ferramentas, os quais este trabalho se propõe a resolver. As vantagens em relação

às ferramentas do tipo JBuilder, IBM VisualAge, Symantec Café e Microsoft Visual J++ são:

- o desenvolvimento da aplicação nessas ferramentas, em última análise, é feito de forma *standalone*. Embora se possa desenvolver clientes e servidores, isto é feito separadamente;
- essas ferramentas utilizam a programação visual apenas para a geração da interface gráfica. Embora em algumas seja possível se desenvolver clientes e servidores, os relacionamentos entre eles e a estrutura global da aplicação distribuída não podem ser visualizados graficamente;
- o fato da aplicação não ser visualizada graficamente (através de um grafo, por exemplo) dificulta a hierarquização do projeto (se não há grafos, também não há subgrafos).

Como pôde ser observado, há basicamente uma vantagem central do trabalho desenvolvido aqui com relação a esses ambientes de programação: a capacidade de desenvolver de maneira integrada uma aplicação distribuída. Por causa disso existe a visualização global da aplicação, seus objetos e relacionamentos.

5.5.2 Desvantagens

Essas ferramentas são destinadas principalmente a aplicações *desktop*. Por isso, possuem capacidades mais importantes nessa área, como geração de interface gráfica e componentes para desenvolvimento de clientes (clientes de bancos de dados, principalmente). Assim, as principais deficiências que a ferramenta proposta apresentam nesse contexto são as seguintes:

- não há suporte para a geração de interface gráfica;
- não há depuração seqüencial: não é possível se depurar o código interno de um objeto;
- a geração de código não é totalmente transparente (há as restrições descritas na seção 4.9.2): se o usuário alterar o código gerado fora da ferramenta, problemas podem ocorrer ao se recarregar o projeto. A ferramenta JBuilder, por exemplo, consegue prover uma transparência muito melhor de geração de código.

As ferramentas mencionadas são muito boas na programação seqüencial. Possuem bons depuradores e compiladores mais completos. A edição dos programas recebe atenção especial e é facilitada com muitos recursos, como realce de sintaxe, complementação de comandos (antes de o usuário terminar de digitar um comando, a ferramenta exibe as possíveis opções), assistentes para criação de aplicações e muitas outras funções. Pela sua importância, estas capacidades podem ser adicionadas futuramente ao trabalho proposto aqui.

5.6 Tabelas comparativas

A fim de melhor apresentar os resultados da comparação, seguem duas tabelas que resumem as principais vantagens e desvantagens deste trabalho com relação às outras soluções mencionadas acima.

TABELA 5.1 - Vantagens.

tipo ferramenta	vantagens com relação à ferramenta correspondente
âmbito geral	- programação orientada a objeto - modelo de componentes de *software* - portabilidade - geração automática de código
ferramentas textuais	- identificação dos relacionamentos entre os objetos - visualização geral da aplicação - a linguagem Java já é conhecida e usada - Java é uma linguagem orientada a objeto - componentes de *software* - eficiência com relação à explicitação da concorrência
ferramentas de programação visual paralela e distribuída	- programação orientada a objeto - suporte a objetos distribuídos - programação orientada a eventos - grande possibilidade de extensão da ferramenta - desenvolvimento modular e reutilização de código - flexibilidade originada da geração de código textual - portabilidade da ferramenta - portabilidade do código gerado - escalabilidade da linguagem visual - execução simultânea da aplicação em diferentes ambientes - inexistência de uma sintaxe de anotação adicional - suporte a diferentes tipos de comunicação - programação cliente-servidor
ferramentas de programação visual em Java	- visualização global da aplicação - desenvolvimento hierárquico das aplicações - desenvolvimento integrado de clientes e servidores

TABELA 5.2 - Desvantagens.

tipo ferramenta	desvantagens com relação à ferramenta correspondente
âmbito geral	- fraco desempenho de Java - necessidade de adaptação do usuário
ferramentas textuais	- as ferramentas textuais possuem mais código desenvolvido - Java é ineficiente - o ambiente de programação é pesado - a explicitação da concorrência é um ônus extra ao usuário - ferramentas textuais são mais simples
ferramentas de programação visual paralela e distribuída	- falta de visualização - falta de depuração - ineficiência de Java para programação paralela - não há suporte direto para paralelismo (de dados, p. ex.) - não suporta características específicas de outras ferramentas, como tolerância a falhas
ferramentas de programação visual em Java	- falta de suporte à geração de interfaces gráficas - não há depuração seqüencial - geração de código não totalmente transparente

6 Conclusões e trabalhos futuros

Este trabalho apresentou o projeto e a implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações que utilizam objetos distribuídos. As principais motivações vieram da própria programação visual, importante em se tratando de ambientes concorrentes, e das carências existentes nas ferramentas atuais de programação visual paralela e distribuída. Estas fazem bom uso da programação visual, mas oferecem suporte limitado a sistemas abertos e aplicações cliente-servidor. Ademais, dificultam o processo de desenvolvimento pela falta na maioria dos casos de orientação a objeto e reutilização de código, entre outras características importantes de engenharia de *software*.

Buscando-se o aperfeiçoamento de tais ferramentas, a solução encontrada foi apresentar um ambiente de desenvolvimento que continuasse a se beneficiar da visualização em programação concorrente, mas que resolvesse os problemas de suporte a sistemas distribuídos e de qualidade de *software*. Para chegar a este objetivo, projetou-se uma ferramenta que combina em um único ambiente vantagens das ferramentas de programação visual para ambientes paralelos e distribuídos e das ferramentas de desenvolvimento rápido de aplicações:

- programação visual;
- programação com objetos distribuídos;
- programação orientada a eventos;
- suporte à reutilização de componentes;
- geração automática parcial de código para a linguagem Java.

Assim, pode-se afirmar que a ferramenta proposta aqui é uma reunião dos principais pontos dos tipos de ferramentas citados com o objetivo de facilitar a tarefa de programação concorrente, principalmente distribuída. Da primeira classe tomou-se a programação visual e a programação distribuída, enquanto das outras vieram os componentes de *software*, a geração de código durante a edição do programa e o modelo da interface gráfica.

Outra forma de se analisar os objetivos da ferramenta é fazê-lo através do prisma da facilidade de uso. Todas as características expostas acima visam a facilitar o desenvolvimento de aplicações distribuídas em Java. Além de dar suporte à geração das aplicações em si, procurou-se criar um ambiente agradável ao usuário, com formato semelhante às ferramentas de maior sucesso no desenvolvimento de aplicações para sistemas *desktop*. Ao final se chegou a um resultado que segundo as análises feitas é capaz de diminuir a complexidade da programação concorrente.

6.1 Resultados

A avaliação da facilidade de uso das ferramentas de programação visual paralela e distribuída é uma questão bastante complicada e ainda pendente [WIL93]. Para quantificar e medir a usabilidade desses sistemas, devem ser levados em conta fatores de interação homem-máquina que comumente são ignorados pela ciência da computação em geral. Várias características determinam a facilidade de uso desses sistemas: curva de aprendizado, erros de programação, desempenho determinístico,

compatibilidade com *software* existente e integração com outras ferramentas. Pouco estudo tem sido feito para avaliar efetivamente a utilidade dessas ferramentas e até que ponto elas realmente auxiliam na programação. Como resultado, a usabilidade de um sistema é medida pela facilidade com que um pequeno conjunto de algoritmos é implementado pelos autores do ambiente. Os critérios dessa avaliação praticamente não existem, tornando o estudo complicado e difícil de validar. Entretanto, os poucos estudos feitos mostraram que, em geral, essas ferramentas são mais amigáveis do que a programação direta numa linguagem de programação textual, principalmente para os usuários que não tiveram muita experiência com programação em qualquer dos ambientes. Embora com algumas deficiências em relação a um método científico de análise, Szafron [SZA94] apresenta um estudo onde conclui que a programação no ambiente visual Enterprise é mais simples do que a programação paralela textual com uma biblioteca de troca de mensagens de baixo nível.

Como foi afirmado, a dificuldade de validação da facilidade de uso ocorre por que não há estudo teórico que possa ser usado para avaliar a facilidade de uso de um ambiente ou metodologia de programação. A conseqüência disso é que se deve realmente implementar e usar os sistemas para compreender os seus méritos e limitações [BRO94]. A constatação acaba sendo feita de forma natural através da aceitação ou não dos usuários, com conclusões freqüentemente subjetivas e dependentes do contexto. De qualquer forma, ainda que não sirva totalmente aos propósitos de todos os tipos de usuários, ser uma opção adicional para parte deles já é um fato importante.

Sendo assim, sem uma implementação, pouco poderia se concluir sobre a validação deste trabalho, o que torna a sua implementação uma atividade obrigatória. Depois disso, para ajudar na validação, esta ferramenta foi utilizada por alunos numa disciplina de objetos distribuídos no Curso de Pós-Graduação do II-UFRGS. Como principais pontos positivos, foram citados pelos seus usuários os seguintes [POR2000]:

- existência de componentes para programação distribuída;
- o modelo de eventos usado facilita a sua utilização para programadores Java tradicionais que não precisam conhecer os mecanismos de passagem de mensagens dos sistemas distribuídos;
- provê uma documentação gráfica do sistema, explicitando as ligações entre os diferentes componentes;
- a interface gráfica é amigável e respeita os padrões dos ambientes de desenvolvimento mais conhecidos atualmente.

Como desvantagens, foi citado principalmente o fato de que a implementação ainda não estava completa. A respeito do modelo de programação, nenhuma crítica foi feita nesta análise.

Embora seja natural pensar que as ferramentas de desenvolvimento mais completas venham a ser largamente usadas, isto não é verdade. Percebe-se que em muitas vezes as ferramentas CASE são abandonadas pouco tempo após a sua introdução nas organizações. A principal causa disto está na questão da curva de aprendizagem. Durante o início do uso da ferramenta, é natural que haja um decréscimo na produtividade em virtude da necessidade que há em se estudar e compreender o seu funcionamento. Com isso, os primeiros trabalhos podem apresentar índices de produtividade menor. Entretanto, com o desenvolvimento de novas aplicações, a fase de aprendizagem já terá passado e a produtividade passa a ser maior. Muitas organizações,

entretanto, não esperam que isto aconteça e em vista dos resultados piores de produtividade desistem logo no início do uso da ferramenta [KEM92].

Apesar das deficiências apresentadas e de opiniões que vão de encontro ao uso dos ambientes de programação visual, os investimentos feitos na pesquisa de novas ferramentas são por si só uma razão para atestar a sua validade. Certamente elas não podem ser consideradas absolutas em programação concorrente, mas as vantagens que possuem em poder representar de maneira mais natural um programa desse tipo são inegáveis. Os recursos de visualização e depuração também são uma tendência e no futuro deverão estar presentes em praticamente todos os ambientes de desenvolvimento, sejam eles visuais ou não.

As ferramentas de programação visual são portanto mais uma opção aos programadores de ambientes concorrentes. Com o passar do tempo e com o avanço da tecnologia, a quantidade e a qualidade das ferramentas será maior, proporcionando a criação de aplicações de maneira mais fácil e produtiva, sem que isso afete significativamente a flexibilidade e o desempenho dos programas.

6.2 Trabalhos Futuros

As ferramentas de programação desse tipo são ambientes complexos e em constante evolução, deixando o seu desenvolvimento sempre sujeito a aprimoramentos e extensões:

- melhoramento da ferramenta para facilitar também a modelagem de aplicações centralizadas, aplicando a programação visual também a esses tipos de aplicações;
- acréscimo de módulos de monitoração e visualização;
- implementação da depuração tanto seqüencial para o código interno do objeto como distribuída para análise da comunicação;
- aprimoramento das facilidades de edição dos programas;
- análise léxica e sintática do código fonte a fim de melhorar a sincronização entre o código textual e o código visual e assim eliminar algumas restrições que existem atualmente na edição do código textual;
- implementação de novos componentes, desde objetos para novos ambientes de execução até outros para o suporte de tipos adicionais de comunicação. Isto pode ser importante para o caso da ferramenta Voyager não ser mais de uso livre e gratuito;
- incorporação de um escalonador para a escolha da máquina mais adequada à instanciação de um novo objeto;
- adição de novos recursos à linguagem visual para aumentar o seu poder de expressão, como comunicação de grupo, espaço de memória global a grafos e objetos replicados;
- suporte a aplicações tolerantes a falhas e a eventos distribuídos;
- integração com ferramentas geradoras de interface gráfica;
- realização de um estudo mais detalhado sobre a usabilidade da ferramenta;

- disponibilização do uso da ferramenta através de uma página na Internet, onde o usuário possa utilizar e fazer testes com a ferramenta sem a necessidade de instalá-la;
- melhoramentos gerais na interface. Conselhos neste sentido podem ser obtidos dos usuários à medida que a ferramenta vai sendo usada;
- geração de aplicações que podem ser integradas mais facilmente às aplicações já existentes;
- inclusão de uma linguagem de *script*, de maneira que o usuário possa construir textualmente a sua aplicação e depois visualizá-la na forma de um grafo.

Esta lista de atividades com certeza pode ser estendida. À medida que o tempo passa e novas tecnologias surgem, necessidades se criam e assim novas características e recursos podem ser implementados.

6.3 Considerações finais

Como resultado último, as contribuições mais importantes deixadas por este trabalho com relação ao que existe atualmente são:

- principal contribuição: projeto e implementação de uma ferramenta visual para programação distribuída onde é possível se visualizar toda a aplicação simultaneamente, com os objetos distribuídos e seus relacionamentos;
- quando considerados individualmente, os conceitos utilizados nessa ferramenta não são totalmente novos. Entretanto, a aplicação deles num único ambiente de desenvolvimento resolve problemas ainda não tratados por outras ferramentas;
- atenção dada aos conceitos de engenharia de *software*: reutilização, orientação a objeto, modularidade e encapsulamento;
- flexibilidade: geração de diferentes tipos de aplicações, com facilidade de edição praticamente irrestrita do código das aplicações.

Resumindo, este trabalho apresentou o projeto e a implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java que, além de auxiliar no desenvolvimento de aplicações distribuídas através da programação visual, procura também oferecer um ambiente que aplique os principais conceitos de engenharia de *software*, como orientação a objeto, modularidade e reutilização. Com base no uso e nos testes feitos, concluiu-se que a ferramenta é capaz de facilitar o desenvolvimento das aplicações, pois oferece ao usuário um modelo de programação flexível e uma interface visual amigável.

Bibliografia

[AHU86] AHUJA, S.; CARRIERO, N.; GELERNTER, D. Linda and Friends. **Computer**, Los Alamitos, CA, v. 19, n. 8, p. 26-34, Aug. 1986.

[AME87] AMERICA, P. H. M. Pool-T: a parallel object-oriented language. In: YONEZAWA, A.; TOKORO, M. (Eds.). **Object-Oriented Concurrent Programming.** Cambridge: MIT Press, 1987.

[AND88] ANDREWS, G. R. et al. An Over-View of the SR Language and Implementation. **ACM Transactions on Programming Languages and Systems**, New York, v. 10, n. 1, p. 51-86, Jan. 1988.

[ARA2000] ARAÚJO, E. B. Uma Ferramenta de Visualização para Aplicações Distribuídas em Java. In: SEMANA ACADÊMICA DO PPGC, 5., 2000. **Anais...** Disponível em: <http://www.inf.ufrgs.br/pos/Semana Academica/Semana2000/EdvarAraujo>. Acesso em: 25 nov. 2000.

[AZE2000] AZEVEDO, S. C. Análise Estática de Programas Orientados a Objetos. SEMANA ACADÊMICA DO PPGC, 5., 2000. **Anais...** Disponível em: <http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana2000/ SilvanaAzevedo>. Acesso em: 25 nov. 2000.

[BAL88] BAL, H. E.; TANENBAUM, A. S. Distributed Programming with Shared Data. In: IEEE CS INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES, 1988, Miami, FL. **Proceedings…** [S.l.:s.n.], 1988.

[BAL90] BAL, H. E.; KAASHOEK, M. F.; TANENBAUM, A. S. Experience with Distributed Programming in Orca. In: IEEE CS INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES, 1990, New Orleans, LA. **Proceedings…** [S.l.:s.n.], 1990.

[BAL91] BAL, H. E. **Programming Distributed Systems**. New Jersey: Prentice Hall International, 1991.

[BAL92] BAL, H. E.; KAASHOEK, M. F.; TANENBAUM, A. S. Orca: A Language for Parallel Programming of Distributed Systems. **IEEE Transactions on Software Engineering**, New York, v. 18, n. 3, p. 190-205, Mar. 1992.

[BAR94] BARCELLOS, A. M. P. et al. Um ambiente para Programação de Aplicações Distribuídas em Redes de Workstations. In: CONGRESSO NACIONAL DE REDES DE COMPUTADORES, 12., Curitiba, 1994. **Anais...** Rio de Janeiro: SBC, 1994.

[BAR95] BARTOLI, A. et al. Graphical Design of Distributed Applications Through Reusable Components. **IEEE Parallel & Distributed Technology**, New York, v. 3, n. 1, p. 37-50, 1995.

[BEG91] BEGUELIN, A. et al. **A User's Guide to PVM Parallel Virtual Machine**. Oak Ridge: Oak Ridge National Laboratory, 1991. (ORNL/TM-11826).

[BEG93] BEGUELIN, A. et al. Visualizing and Debugging in a Heterogeneous Environment. **Computer**, Los Alamitos, CA, v. 26, n. 6, p. 88-95, June 1993.

[BEG94] BEGUELIN, A. et al. **HeNCE**: A Users' Guide Version 2.0**.** 1994. Disponível em: <ftp://netlib2.cs.utk.edu/hence/HeNCE-2.0-doc.ps.gz>. Acesso em: 20 nov. 1998.

[BOR2000] BORLAND. **Delphi**. Disponível em: <http://www.borland.com/delphi>. Acesso em: 25 nov. 2000.

[BOR2000a] BORLAND. **C++ Builder**. Disponível em: <http://www.borland.com/bcppbuilder>. Acesso em: 25 nov. 2000.

[BOR2000b] BORLAND. **JBuilder**. Disponível em: <http://www.borland.com/jbuilder>. Acesso em: 25 nov. 2000.

[BRI98] BRIOT, J.-P.; GUERRAOUI, R.; LÖHR, K.-P. Concurrency and Distribution in Object-Oriented Programming. **ACM Computing Surveys**, New York, v. 30, n. 3, p. 291-329, Sept. 1998.

[BRO94] BROWNE, J. C. et al. **Visual Programming and Parallel Computing.** Austin: Dept. of Computer Sciences, Univ. of Texas at Austin, 1994. (TR94-229). Disponível em: <ftp://ftp.cs.utexas.edu/pub/code2/ut-cs-94-229.ps.Z>. Acesso em: 25 nov. 2000.

[BRO98] BROWN, A. W.; WALLNAU, K. C. The Current State of CBSE**. IEEE Software**, Los Alamitos, CA, v. 15, n. 5, p. 37-46, Sept./Oct. 1998.

[BUR95] BURNETT, M. et al. Scaling Up Visual Programming Languages. **Computer**, Los Alamitos, CA, v. 18, n. 3, p. 45-54, Mar. 1995.

[BUR95a] BURNETT, M. M.; McINTYRE, D. W. Visual Programming**. Computer**, Los Alamitos, CA, v. 18, n. 3, p. 14-16, Mar. 1995.

[CAI95] CAI, W.; PIAN, T. L.; TURNER, S. J. A Framework for Visual Parallel Programming. In: AIZU INTERNATIONAL SYMPOSIUM ON PARALLEL ALGORITHMS/ARCHITECTURE SYNTHESIS, 1995, Japan. **Proceedings…** Japan: IEEE Computer Society Press, 1995.

[CAR89] CARRIERO, N.; GELERNTER, D. How to Write Parallel Programs: a Guide to the Perplexed. **ACM Computing Surveys**, New York, v. 21, n. 3, p. 323-357, Sept. 1989.

[CAR89a] CARRIERO, N.; GELERNTER, D. Linda in Context. **Communications of the ACM**, New York, v. 32, n. 4, p. 444-458, Apr. 1989.

[CAR93] CAROMEL, D. Towards a method of object-oriented concurrent programming**. Communications of the ACM**, New York, v. 36, n. 9, p. 90-102, Sept. 1993.

[CAR95] CARD, D. N. The RAD Fad: Is Timing Really Everything? **IEEE Software**, Los Alamitos, CA, v. 12, n. 5, p. 19-22, Sept. 1995.

[CAV93] CAVALHEIRO, G, G. H.; NAVAUX, P. O. A. DPC++: Uma Linguagem para Processamento Distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES – PROCESSAMENTO DE ALTO DESEMPENHO, SBAC-PAD, 5., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993.

[CHA91] CHAN, E. et al. **Enterprise:** an interactive graphical programming environment for distributed software development. Edmonton, Canada: University of Alberta, 1991. (TR 91-17).

[COL98] COLBY, C. et al. **Objects and Concurrency in Triveni**: A Telecommunication Case Study in Java. 1998. Disponível em: < http://www.cs.luc.edu/triveni/papers/coots98.ps.gz>. Acesso em: 25 nov. 2000.

[COL98a] COLBY, C. et al. **Design and Implementation of Triveni**: a Process-algebraic API for Threads + Events. 1998. Disponível em: < http://www.cs.luc.edu/triveni/papers/iccl98.ps.gz>. Acesso em: 25 nov. 2000.

[COR98] CORNELL, G. **Core Java**. São Paulo: Makron Books, 1998. 807p.

[COX98] COX, P. T.; GLASER, H.; MACLEAN, S. **A Visual Development Environment for Parallel Applications**. 1998. Disponível em: <http://cui.unige.ch/Visual/local/CoxGlaserMaclean99.ps.gz>. Acesso em: 25 nov. 2000.

[CUG98] CUGOLA, G.; NITTO, E. D.; FUGGETTA, A. **Exploiting and event-based infrastructure to develop complex distributed systems**. Milano, Italy: CEFRIEL - Politecnico di Milano, 1998. Disponível em: <http://www.acm.org/pubs/articles/proceedings/soft/302163/p261-cugola/p261-cugola.pdf>. Acesso em: 25 nov. 2000.

[DAV96] DAVOLI, R. et al. Parallel Computing in Networks of Workstations with Paralex. **IEEE Transactions on Parallel and Distributed Systems**, New York, v. 7, n. 4, p. 371-384, Apr. 1996.

[DIA71] DIAS, D. S.; LUCENA, A. J. P. L.; FARIA, L. **Programação FORTRAN**. Rio de Janeiro: Ao Livro Técnico, 1971. 257 p.

[FAR98] FARLEY, J. **Java Distributed Computing**. USA: O'Reilly & Associates, 1998.

[FEL94] FELIX, C. A.; COSTA, C. M. **Sistemas de Memória Compartilhada Distribuída:** trabalho individual. Porto Alegre: CPGCC da UFRGS, 1994. 53 p. (TI-422).

[FER2000] FERRARI, D. N; VARGAS, P. K.; GEYER, C. F. R. ReMMoS - um modelo de replicação em ambientes que permitem mobilidade de objetos. In: WORKSHOP DE SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2000, São Pedro, SP, Brasil. **Anais...** São Carlos: UFSCar, 2000.

[FIS90] FISHER, A. S. **CASE - Utilização de Ferramentas para Desenvolvimento de Software.** Rio de Janeiro: Campus, 1990.

[FOW97] FOWLER, M.; SCOTT, K. **UML Distilled - Applying the Standard Ibject Modeling Language.** New York: Addison-Wesley Longman, 1997.

[GLA99] GLASS, G. **Overview of Voyager:** ObjectSpace's Product Family for State-of-the-Art Distributed Computing**.** Dallas: ObjectSpace, 1999.

[GLA99a] GLASS, G. **Reducing Development Effort using the ObjectSpace Voyager ORB.** Dallas: ObjectSpace, 1999.

[GLI84] GLINERT, E. P.; TANIMOTO, S. L. PICT: An interactive graphical programming environment**. Computer**, Los Alamitos, CA, v. 17, n. 11, p. 7-28, Nov. 1984.

[GRU95] GRUNDY, JOHN et al. Connecting the Pieces**.** In: BURNETT, M. M.; GOLDBERG, A.; LEWIS, T. G. **Visual Object-Oriented Programming - Concepts and Environments**. Greenwich, CT: Manning Publications, 1995.

[GUE99] GUERRAOUI, R.; FAYAD, M. E. OO Distributed Programming Is *Not* Distributed OO Programming**. Communications of the ACM**, New York, v. 42, n. 4, p. 101-104, Apr. 1999.

[HEL90] HELLER, D. **XView Programming Manual - Volume Seven.** Sebastopol: O'Reilly & Associates, 1990.

[HEL91] HELLER, D. **Motif Programming Manual.** Sebastopol: O'Reilly & Associates, Inc, 1991. (The X Window System Series, v. 6).

[HER98] HERICKO, M. et al. Java and Distributed Object Models: An Analysis. **ACM Sigplan Notices**, New York, v. 33, n. 12, p. 57-65, Dec. 1998.

[HIR9?] HIRANO, S.; YASU, Y.; IGARASHI, H. **Performance Evaluation of Popular Distributed Object Technologies for Java**. [199?]. Disponível em: <http://www.kav.cas.cz/koi/~buble/corba/others/PerfEvalJava. HORB.pdf>. Acesso em: 25 nov. 2000.

[HOL98] HOLZNER, S. **Dominando o Visual J++**. São Paulo: Makron Books, 1998. 550 p.

[IBM2000] IBM. **IBM Aglets Software Development Kit - Home Page**. Disponível em: <http://www.trl.ibm.co.jp/aglets/>. Acesso em: 25 nov. 2000.

[KAC96] KACSUK, P.; J.; DÓZSA, G.; FADGYAS. **Designing Parallel Programs by the Graphical Language GRAPNEL**. 1996. Disponível em: <ftp://ftp.lpds.sztaki.hu/pub/lpds/publications/GRADE/grapnel-journal96.ps.gz>. Acesso em: 25 nov. 2000.

[KAC97] KACSUK, P. **Graphical environments to support parallel program development.** 1997. Disponível em: <ftp://ftp.lpds.sztaki.hu/pub/lpds/publications/GRADE/zakopane97.ps.gz>. Acesso em: 25 nov. 2000.

[KAC97a] KACSUK, P. et al. A graphical development and debugging environment for parallel programs. **Parallel Computing Journal**, Amsterdam, v. 22, n. 13, p. 1747-1770, Feb. 1997.

[KER78] KERNIGHAN, B. W.; RITCHIE, D. M **The C Programming Language.** Englewood Cliffs: Prentice Hall, 1978. 228 p.

[KEM92] KEMERER, C. F. How the Learning Curve Affects CASE Tool Adoption. **IEEE Software**, Los Alamitos, CA, v. 9, n. 3, p. 23-28, May 1992.

[KLA99] KLABUNDE, C. C. **Um Estudo sobre Componentes de Software**: trabalho individual. Porto Alegre: CPGCC da UFRGS, 1999. (TI-757).

[KOZ98] KOZACZYNSKI, W.; BOOCH, G. Component-Based Software Engineering. **IEEE Software**, Los Alamitos, CA, v. 15, n. 5, p. 34-36, Sept./Oct. 1998.

[KRI95] KRISHNAMURTHY, B.; ROSENBLUM, D. S. Yeast: A General Purpose Event-Action System. **IEEE Transactions on Software Engineering**, New York, v. 21, n. 10, p. 845-857, Oct. 1995.

[LAW2000] LAWSON, S. **Java™ Integrated Programming Environment.** Disponível em: <http://e-i-s.co.uk/jipe>. Acesso em: 25 nov. 2000.

[LOQ98] LOQUES, O.; LEITE, J.; CARRERA, E. V. P-RIO: A Modular Parallel-Programming Environment. **IEEE Concurrency**, Los Alamitos, CA, v. 6, n. 1, p. 47-57, Jan.-Mar. 1998.

[MAL97] MALACARNE, J. **Implementação de um Ambiente Gráfico para o Desenvolvimento de Aplicações Distribuídas**: projeto de diplomação. Porto Alegre: CIC da UFRGS, 1997. 91 p.

[MAL99] MALACARNE, J. **Ambientes de Programação Visual Paralela e Distribuída**: trabalho individual. Porto Alegre: PPGC da UFRGS, 1999. 91 p. (TI-776).

[MAR99] MARTINS, A. A. **Um estudo sobre o desenvolvimento de aplicações Java utilizando o IBM VisualAge for Java e o AS/400**: projeto de diplomação. Porto Alegre: CIC da UFRGS, 1999, 62 p.

[MIC2000] MICROSOFT TECHNET. **DCOM –** The Distributed Component Object Model**.** Disponível em: <http://www.microsoft.com/TechNet/winnt/Winntas/technote/dcomwp.asp>. Acesso em: 25 nov. 2000.

[MIT2000] MITSUBISH. **Concordia -** Java Mobile Agent Technology. Disponível em: <http://www.meitca.com/HSL/Projects/Concordia/>. Acesso em: 25 nov. 2000.

[MPI94] MESSAGE PASSING INTERFACE FORUM. MPI: A Message-Passing Interface Standard. **Journal of Supercomputing Applications**, [S.l], v. 8, n. 3/4, 1994.

[NEW92] NEWTON, P.; BROWNE, J. C. **The CODE 2.0 Graphical Parallel Programming Language.** 1992. Disponível em: <ftp://ftp.cs.utexas.edu/pub/code2/CodeICS92.ps.Z>. Acesso em: 25 nov. 2000.

[NEW93] NEWTON, P.; KHEDEKAR, S. Y. **CODE 2.0 User and Reference Manual**. Austin: Univ. of Texas at Austin, 1993. Disponível em: <http://www.cs.utexas.edu/users/code/download/docs/CODE_2.0_Manual.ps.gz>. Acesso em: 25 nov. 2000.

[NEW95] NEWTON, P.; DONGARRA, J. **Overview of VPE:** A Visual Environment for Message-Passing**.** 1995. Disponível em: <ftp://cs.utk.edu/pub/newton/vpe/docs/hcw95.ps.Z>. Acesso em: 25 nov. 2000.

[OMG98] OMG - OBJECT MANAGEMENT GROUP. **CORBA Services.** 1998. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/98-12-09.pdf>. Acesso em: 25 nov. 2000.

[OMG2000] OMG - OBJECT MANAGEMENT GROUP. **CORBA -** Common Object Request Broker Architecture**.** Disponível em: <http://www.omg.org/>. Acesso em: 25 nov. 2000.

[OMG2000a] OMG - OBJECT MANAGEMENT GROUP. **Unified Modeling Language (UML) 1.3 specification.** 2000. Disponível em: <ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf>. Acesso em: 25 nov. 2000.

[PAN93] PANDEY, R.; BROWNE, J. C. Event-based Composition of Concurrent Programs. In: INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, 6., 1993, Portland, Oregon. **Proceedings…** Berlin: Springer-Verlag, 1993. p. 115-138. (Lecture Notes in Computer Science, v. 768).

[PAT2000] PATWARDHAN, B. **Event Driven Programming Paradigm**. Disponível em: <http://www.ncst.ernet.in/~guifac/bipin/paradigm/paradigm.html>. Acesso em: 25 nov. 2000.

[PED99] PEDERSEN, J. B.; WAGNER, A. PVMbuilder - A Tool for Parallel Programming. In: EURO-PAR, 1999, Toulouse, France. **Proceedings…**

Berlin: Springer-Verlag, 1999. p. 108-112. (Lecture Notes in Computer Science, v. 1685).

[PET98] PETROUTSOS, E. **Dominando o Visual Basic 5**. São Paulo: Makron Books, 1998. 919 p.

[PGI2000] PORTLAND GROUP. **The Portland Group -** Parallelizing Compilers and Development Tools. Disponível em: <http://www.pgroup.com>. Acesso em: 25 nov. 2000.

[PIC96] PICTORIUS. **Prograph**. Disponível em: < http://www.pictorius.com/prograph.html>. Acesso em: 25 nov. 2000.

[POR2000] PORTO, E. K. K.; PILLA, M. L. **DOBuilder**. Disponível em: <http://www.inf.ufrgs.br/procpar/disc/cmp167/trabalhos/sem2000-1/T2/kenzo_pilla/>. Acesso em: 25 nov. 2000.

[PUR98] PURAO, S.; JAIN, H. NAZARETH, D. Effective Distribution of Object-Oriented Applications. **Communications of the ACM**, New York, v. 41, n. 8, p. 100-109. Aug., 1998.

[QUE90] QUERCIA, V.; O'REILLY, T. **X Window System User´s Guide**. Sebastopol: O'Reilly & Associates, 1990. 709 p.

[SCH96] SCHRAMM, J. F. L. **Ambiente Gráfico para o Desenvolvimento de Aplicações Distribuídas.** Porto Alegre: CPGCC da UFRGS, 1996. 78 p. Dissertação de Mestrado.

[SCH96a] SCHMID, H. A. Creating Applications from Components: A Manufacturing Framework Design. **IEEE Software**, Los Alamitos, CA, v. 13, n. 6, p. 67-75, Nov. 1996.

[SCH98] SCHILDT, H. **Windows 98 Programming from the Ground Up**. Berlekey: McGraw-Hill, 1998. 809 p.

[SIN97] SINHA, P. K. **Distributed Operating System**. New York: IEEE Computer Society, 1997. 743 p.

[SIV94] SIVILOTTI, P. A. G.; CARLIN, P. A. **A Tutorial for CC++**. Pasadena, CA: Department of Computer Science, California Institute of Technology, 1994. Disponível em: <http://globus.isi.edu/ccpp>. Acesso em: 25 nov. 2000.

[STA94] STARINGER, W. Constructing Applications form Reusable Components. **IEEE Software**, Los Alamitos, CA, v. 11, n. 5, p. 61-68, Sept. 1994.

[STA95] STAROVIC, G.; CAHILL, V.; TANGNEY, B. **An Event Based Object Model for Distributed Programming**. Dublin, Ireland: Distributed Systems Group, Department of Computer Science, Trinity College, 1995. Disponível em: <ftp://ftp.cs.tcd.ie/pub/tech-reports/reports.95/TCD-CS-95-28.ps.gz>. Acesso em: 25 nov. 2000.

[SUN2000] SUN MICROSYSTEMS. **Jini™ Technology Core Platform Specification.** 2000. Disponível em: <http://www.sun.com/jini/specs/core1_1.pdf>. Acesso em: 25 nov. 2000.

[SUN2000a] SUN MICROSYSTEMS. **JavaBeans.** Disponível em: <http://java.sun.com/products/jdk/rmi/>. Acesso em: 25 nov. 2000.

[SUN2000b] SUN MICROSYSTEMS. **RMI.** Disponível em: <http://java.sun.com/products/jdk/rmi/>. Acesso em: 25 nov. 2000.

[SUN2000c] SUN MICROSYSTEMS. **Javadoc Tool Homepage.** Disponível em: <http://java.sun.com/products/jdk/javadoc/>. Acesso em: 25 nov. 2000.

[SUN2000d] SUN MICROSYSTEMS. **The Beans Development Kit.** Disponível em: <http://java.sun.com/docs/books/tutorial/javabeans/whatis/bdkDescription.html>. Acesso em: 25 nov. 2000.

[SUN2000e] SUN MICROSYSTEMS. **The Java Tutorial.** Disponível em: <http://java.sun.com/docs/books/tutorial/index.html>. Acesso em: 25 nov. 2000.

[SUN2000f] SUN MICROSYSTEMS. **Java HotSpot™ Technology.** Disponível em: <http://java.sun.com/products/hotspot/>. Acesso em: 25 nov. 2000.

[SZA94] SZAFRON, D.; SCHAEFFER, J. **An Experiment to Measure the Usability of Parallel Programming Systems**. 1994. Disponível em: <http://www.cs.ualberta.ca/~enter/TechReports/TR94-03.ps>. Acesso em: 18 jan. 1999.

[TAN96] TANENBAUM, A. S. **Computer Networks**. 3rd ed. New Jersey: Prentice Hall PTR, 1996.

[TAN97] TANENBAUM, A. S.; WOODHULL, A. S. **Operating Systems - Design and Implementation**. 2nd ed. New Jersey: Prentice Hall PTR, 1997.

[TRA97] TRACZ, W. Developing Reusable Java Componentes. **Software Engineering Notes**, New York, v. 22, n. 3, May, 1997.

[VOA98] VOAS, J. Maintaining Component-Based Systems. **IEEE Software**, Los Alamitos, CA, v. 15, n. 4, p. 22-27, July/Aug. 1998.

[WAL94] WALDO, J. et al. **A Note on Distributed Computing**. Mountain View, CA: Sun Microsystems Laboratories Inc, 1994. (SMLI TR-94-29). Disponível em: <http://www.sun.com/research/techrep/1994/abstract-29.html>. Acesso em: 25 nov. 2000.

[WEY98] WEYUKER, E. J. Testing Component-Based Software: A Cautionary Tale. **IEEE Software**, Los Alamitos, CA, v. 15, n. 5, p. 54-59, Sept./Oct. 1998.

[WIL93] WILSON, G. V.; SCHAEFFER, J.; SZAFRON, D. **Enterprise in Context:** Assessing the Usability of Parallel Programming Environments. Edmonton, Canada: Department of Computer Science. The University of Alberta, 1993. (TR 93-09). Disponível em: <http://www.cs.ualberta.ca/~enter/TechReports/TR93-09.ps>. Acesso em: 18 jan. 1999.

[WIR94] WIRTZ, G. **The Meander Language and Programming Environment.** 1994. Disponível em: <ftp://ftp.informatik.uni-siegen.de/pub/papers/meander/SMSTPE-Sept94-moscow.ps.gz>. Acesso em: 25 nov. 2000.

[WIR94a] WIRTZ, G. **Graph-Based Software Construction for Parallel Message-Passing Programs**. 1994. Disponível em: <http://wwwmath.uni-muenster.de/math/inst/info/Service/AGs/versys/research/meander/PAPERS/InfSoftTech94.ps.gz>. Acesso em: 25 nov. 2000.

[WIR94b] WIRTZ, G. **Modularization and Process Replication in a Visual Parallel Programming Language.** 1994. Disponível em: <ftp://ftp.informatik.uni-siegen.de/pub/papers/meander/IEEE-VisLangSymp-Sept94-StLouis.ps.gz>. Acesso em: 25 nov. 2000.

[YON87] YONEZAWA, A.; TOKORO, M. (Eds). **Object-Oriented Concurrent Programming.** Cambridge: MIT Press, 1987.

[ZER92] ZERNIK, D.; SNIR, M.; MALKI, D. Using Visualization Tools to Understand Concurrency. **IEEE Software**, Los Alamitos, CA, v. 9, n. 3, p. 87-92, May 1992.

[ZHA94] ZHANG, K.; MA, W. **Graphical Assistance in Parallel Program Development.** 1994. Disponível em: <ftp://ftp.mpce.mq.edu.au/pub/comp/papers/zhang.VL94.ps.Z>. Acesso em: 25 nov. 2000.