

Roque Maitino Neto

*Visualização Tridimensional de Programas
Orientados a Objeto*

Marília

Outubro de 2006

Roque Maitino Neto

*Visualização Tridimensional de Programas
Orientados a Objeto*

Dissertação apresentada ao Programa de
Mestrado em Ciência da Computação da
Fundação Eurípides Soares da Rocha de
Marília como requisito para obtenção do
título de Mestre em Ciência da Computação.

Orientador:

Prof. Dr. Márcio Eduardo Delamaro

Co-orientadora:

Prof. Dra. Fátima de Lourdes dos Santos Nunes

MESTRADO EM CIÊNCIA DA COMPUTAÇÃO
PPGCC - PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
FUNDAÇÃO EURÍPIDES SOARES DA ROCHA

Marília

Outubro de 2006

*Dedico esta dissertação a meus pais,
meus exemplos mais luminosos de sobriedade
e persistência.*

Agradecimentos

Dedico meus sinceros agradecimentos para:

- o Prof. Dr. Márcio Eduardo Delamaro, pela orientação e incentivo;
- a Prof^a. Dr^a. Fátima de Lourdes dos Santos Nunes, pela orientação e auxílio na conclusão deste trabalho;
- ao amigo Sérgio Roberto Delfino, por compartilhar parte de seu conhecimento em Java e Java3D, colaborando assim de maneira inestimável na implementação deste projeto. Agradeço de forma especial pelo amparo técnico nos momentos de dificuldade e pela paciência e presteza com que dispensou tal auxílio.
- a Fundação de Ensino Eurípides Soares da Rocha de Marília e
- todos os colegas do Mestrado em Ciência da Computação da Fundação de Ensino Eurípides Soares da Rocha de Marília.

*”Não tem sentido dizer que fazemos o melhor que podemos.
Temos de conseguir fazer o que é necessário.”*

Winston Churchill

Resumo

Em conseqüência de uma crescente demanda – exercida tanto pela comunidade científica quanto por leigos – por formas mais apropriadas de apresentação de imagens que representem um ambiente particular, inúmeros estudiosos têm se empenhado em transportar para o universo visual certas realidades outrora confinadas apenas na concepção de poucos. O presente trabalho objetiva a apresentação de uma ferramenta de visualização de software orientado a objeto. Através do reconhecimento dos componentes de um programa Java, ela é capaz de exibi-los por meio de metáforas tridimensionais, facilitando a interação entre o usuário e os objetos exibidos. Compõe ainda seu escopo a apresentação de conceitos relacionados à visualização de software orientado a objeto, principalmente por meio da apresentação de pesquisas e trabalhos relacionados. São colocados em foco aspectos da visualização de dados (incluindo os de natureza científica) e sua aplicabilidade na educação, além de conceitos concernentes ao espaço tridimensional. São apresentados exemplos de mapeamentos de dados originais em metáforas, destacando a importância deste procedimento na obtenção de uma visualização de compreensão satisfatória e imediata. A expressividade e a efetividade assumem papéis de elementos de avaliação destas metáforas. O estudo da programação orientada a objeto revela as estruturas de programa baseado neste paradigma que são apresentadas através de metáforas na efetiva implementação do projeto. Em sua parte final, o trabalho apresenta descrições relacionadas ao uso da ferramenta, examina resultados de aplicações da ferramenta em alguns programas e propõe trabalhos futuros como desdobramento e evolução do projeto nesta oportunidade apresentada.

Palavras-chave: Visualização de software, visualização da informação, visualização de programas orientados a objeto.

Abstract

Due to an increasing demand – carried out even by the scientific community or by the laymen – to more appropriate forms of image presentation that represent a particular environment, many researchers have been engaging themselves in transporting to the visual universe realities formerly confined in the conception of a few people. The aim of this work is the implementation of a object-oriented software visualization tool. Through the recognition of the components of a Java program, it is capable to exhibit them through three-dimensional metaphors, making easier interactions between the user and the exhibited objects. Its scope includes the approach of concepts related to object-oriented software visualization, mainly through the presentation of related researches and works. Aspects of the data visualization are placed in focus (including the scientific nature ones) and its applicability in education, besides concepts related to three-dimensional space. Examples of original data mapping in metaphors are presented, highlighting the importance of this activity in the acquisition of a visualization of as immediate and satisfactory comprehension. The expressivity and the effectiveness assume roles of elements of evaluation of these metaphors. The research of object oriented programs theory reveals the program structures based on this paradigm that should be presented through metaphors in the effective implementation of the project. In its final section, this work presents descriptions related to the use of tool, discusses the application of tool in some programs and proposes future work as evolution of the work presented.

Keywords: Software visualization, information visualization, object oriented software visualization.

Lista de Figuras

1	Exemplo de interface de visualização médica	p. 23
2	Visualização de neuroimagem	p. 24
3	Sistema Lifelines	p. 25
4	Visão do protótipo de ambiente colaborativo de geovisualização	p. 26
5	Uma janela de visualização da ferramenta <i>WorldWatcher</i>	p. 30
6	Modelo de referência para visualização	p. 34
7	Os eixos do espaço tridimensional	p. 39
8	Transparência no espaço 3D	p. 39
9	Transparência no espaço 2D	p. 39
10	Efeito de profundidade no espaço 3D	p. 40
11	Efeito de profundidade no espaço 2D	p. 40
12	Metáfora de cidade 3D	p. 41
13	A cidade tridimensional na visão aérea	p. 43
14	Vista aproximada da estrutura de um software orientado a objetos	p. 44
15	Diagrama de objetos de um programa em visualização tridimensional	p. 44
16	Duas abordagens para representar formas: não encapsulada e encapsulada	p. 50
17	Exemplo de criação de duas instâncias de uma classe	p. 52
18	Exemplo de criação de subclasse	p. 53
19	Exemplo de criação de nova classe	p. 53
20	Exemplo de criação de uma subclasse	p. 54
21	Exemplo de herança no paradigma OO.	p. 55
22	Exemplificação de polimorfismo	p. 56

23	Exemplo de classe Java contendo métodos e atributos	p. 58
24	Diagrama de classes da ferramenta	p. 61
25	Exemplo de grafo de cena	p. 63
26	Grafo de cena de uma das classes da ferramenta	p. 64
27	Hierarquia entre bibliotecas implementadas na ferramenta do projeto .	p. 65
28	Metáforas utilizadas na ferramenta	p. 66
29	Execução inicial da ferramenta	p. 68
30	Detalhamento dos métodos da classe <code>ClassExhi.java</code>	p. 69
31	Detalhamento dos atributos da classe <code>ClassExhi.java</code>	p. 70
32	Assinatura do método <code>exibeClasses</code>	p. 70
33	Tipo do atributo <code>argumento</code>	p. 71
34	Detalhe de objetos que representam agregação	p. 71
35	Detalhamento dos atributos da classe <code>Generic</code> , a partir do cubo da agregação	p. 72
36	Execução inicial da ferramanta	p. 73
37	Aplicação de zoom na visualização das classes	p. 74
38	Aplicação de rotação na visualização das classes	p. 74
39	Pacotes importados para a classe <code>ClassIden.java</code>	p. 74
40	Código-fonte da classe <code>ClassIden.java</code>	p. 75
41	Pacotes importados na classe <code>ClassExhi.java</code>	p. 76
42	Construtor da classe <code>ClassExhi.java</code>	p. 77
43	Primeira parte do método <code>exibeClasses()</code> , da classe <code>ClassExhi.java</code>	p. 78
44	Segunda parte do método <code>exibeClasses()</code> , da classe <code>ClassExhi.java</code>	p. 79
45	Início do método <code>criaGrafoDeCena()</code> , da classe <code>ClassExhi.java</code> . .	p. 79
46	Parte final do método <code>criaGrafoDeCena()</code> , da classe <code>ClassExhi.java</code>	p. 80
47	Classe <code>EventoC()</code>	p. 81

48	Classe EventoR()	p.81
49	Método criaGrafoDeCenaH(), da classe MethodExhi.java	p.82
50	Árvore hierárquica da classe ClassExhi.java	p.83
51	Método exhibeCampos(), da classe FieldsExhi.java	p.84
52	Método criaGrafoDeCenaF(), da classe FieldsExhi.java	p.84
53	Execução inicial da ferramenta em análise ao programa Editor.java .	p.87
54	Detalhamento dos métodos da classe Editor.java, com aplicação de zoom na parte 2	p.88
55	Diagrama de classes de JavaKitTest.java	p.89
56	Execução inicial da ferramenta aplicada ao programa JavaKitTest.java	p.90
57	Demais classes de JavaKitTest.java	p.91
58	Rotação aplicada a visualização do programa JavaKitTest.java	p.92

Lista de abreviaturas e siglas

2D Bidimensional

3D Tridimensional

API Application Programming Interface

BCEL Byte Code Engineering Library

CAD Computer Aided Design

COOL Language for Comprehending Object Oriented Software

EDA Exploratory Data Analysis

GIS Geographic Information Systems

GVis Geographic Visualization

ITS Intelligent Tutoring Systems

JaBUTi Java Bytecode Understanding and Testing

JVM Java Virtual Machine

MIT Massachusetts Institute of Technology

OO Orientado a Objeto

POO Programação Orientada a Objeto

SSciVEE Supportive Scientific Visualization Environments for Education

Lista de Tabelas

1	Visualização da informação comparada com a visualização científica. . .	p. 21
2	Descrição das entidades no sistema COOL	p. 37

Sumário

1	Introdução	p. 15
1.1	Objetivo	p. 16
1.2	Motivações	p. 16
1.3	Organização	p. 17
2	Visualização de dados	p. 19
2.1	Definição	p. 20
2.2	Tipos de visualização	p. 21
2.2.1	Visualização da informação na medicina	p. 22
2.2.2	Visualização geográfica	p. 25
2.2.3	Visualização científica	p. 27
2.2.3.1	Aplicação da visualização científica na educação	p. 28
2.3	Aspectos relevantes no projeto de uma aplicação de visualização	p. 29
2.4	Visualização de software	p. 32
2.4.1	Visualização de software orientado a objeto	p. 36
2.5	Visualização tridimensional	p. 38
2.5.1	Fundamentos de ambientes 3D	p. 38
2.6	Metáforas	p. 40
2.6.1	Exemplos de metáforas	p. 41
3	Programação orientada a objeto	p. 46
3.1	Conceitos	p. 47

3.1.1	Objetos	p. 47
3.1.2	Abstração	p. 48
3.1.3	Encapsulamento	p. 48
3.1.4	Ocultação de informação	p. 50
3.1.5	Classes	p. 52
3.1.6	Subclasses	p. 53
3.1.7	Reutilização de classes	p. 54
	3.1.7.1 Agregação ou composição	p. 54
	3.1.7.2 Herança	p. 55
3.1.8	Polimorfismo	p. 55
3.1.9	Métodos	p. 56
3.1.10	Campos	p. 58
	3.1.10.1 Modificadores de campo	p. 59

4	Ferramenta de Visualização Tridimensional de Programas Orientados a Objeto	p. 60
4.1	Tecnologias utilizadas	p. 60
	4.1.1 API Java 3D	p. 62
	4.1.2 Pacote lookup	p. 64
	4.1.3 Biblioteca BCEL	p. 65
4.2	Metáforas utilizadas	p. 66
4.3	Aspectos operacionais	p. 67
	4.3.1 Interação com usuário	p. 68
4.4	Aspectos de implementação	p. 73
	4.4.1 Classe <code>ClassIden.java</code>	p. 74
	4.4.2 Classe <code>ClassExhi.java</code>	p. 75
	4.4.3 Classe <code>MethodExhi.java</code>	p. 80

4.4.4	Classe <code>FieldsExhi.java</code>	p.83
4.4.5	Classe <code>Generic.java</code>	p.85
4.4.6	Classes <code>FieldsInsideSphere.java</code> e <code>MethodsInsideSphere.java</code>	p.85
5	Resultados e discussões	p.86
5.1	Programa <code>Editor.java</code>	p.86
5.2	Programa <code>JavaKitTest.java</code>	p.88
5.3	Avaliação dos resultados	p.91
6	Considerações Finais	p.95
	Referências	p.97

1 *Introdução*

Seres humanos são criaturas visuais, em sua essência (RUSS, 2002). Para receber a quase totalidade da informação que nos cerca, o homem depende da sua visão e do correto processamento daquilo que ela incute. Ao contrário da espécie humana, muitos animais dependem menos de suas faculdades visuais e mais de outros sentidos para sobreviverem.

A informação visual – que, por si, é bem assimilada pelos seres humanos – quando bem organizada e disposta de forma estruturada, eleva nossos níveis de compreensão sobre a informação recebida. Tal particularidade humana se faz mais evidente quando a informação é apresentada em formato gráfico. O uso de cores, formas e texturas – entre outras técnicas – torna a compreensão de certa informação bem mais efetiva, em oposição ao uso exclusivo de texto.

A visão humana é primariamente qualitativa e comparativa, ao invés de ser quantitativa. Avaliamos o tamanho e a forma dos objetos pela rotação e sobreposição mental dos mesmos, para que, em seguida, possamos fazer uma comparação direta (RUSS, 2002).

No mundo computacional, tais verdades são ainda mais sólidas. A real correspondência entre a informação e sua visualização é meta para profissionais comprometidos com a representação gráfica de dados através do computador. As mais recentes ferramentas de visualização desenvolvidas (PANAS; BERRIGAN; GRUNDY, 2003; BALZER; NOACK; DEUSSEN, 2004) fazem uso extensivo de elementos gráficos, na clara intenção de tirar vantagem de nossa percepção visual superior e da melhor compreensão sobre a informação que certos elementos visuais são capazes de proporcionar. Assim, pode-se dizer que a visualização une dois dos mais poderosos sistemas de processamento da informação conhecidos: a mente humana e os computadores modernos.

Uma grande variedade de informações em formato gráfico faz parte de nosso cotidiano. Mapas de previsão do tempo, demonstrativos de receitas e despesas e gráficos representando os níveis de inflação são exemplos de meios eficientes de apresentação de dados.

Cientistas, engenheiros, médicos, analistas de negócios, entre outros profissionais, ne-

cessitam analisar com frequência grande quantidade de dados. É comum que, no processamento destes dados, sejam gerados arquivos extensos. A atividade de análise das tendências e relações geralmente é uma tarefa tediosa. No entanto, se os dados forem convertidos para uma forma visual, as tendências e padrões das informações podem ficar imediatamente aparentes (HEARN; BAKER, 1997).

Embora tenha sua importância resguardada, a representação desse tipo de informação não é a única vantagem que o desenvolvimento das novas formas de visualização oferece. A visualização pode ser uma ferramenta útil em muitas áreas do ensino da computação. Estudantes freqüentemente têm dificuldades em assimilar conceitos novos de programação e outros conteúdos. A utilização de métodos visuais pode ser um meio efetivo de aprendizagem e a visualização de classes, objetos e métodos e pode constituir uma importante aliada no processo de apresentação do paradigma de orientação a objetos (DERSHEM; VANDERHYDE, 1998). Da mesma forma, a compreensão da complexidade cada vez maior dos sistemas computacionais pode ser auxiliada por meio da exibição de objetos no espaço tridimensional (WARE; HUI; FRANCK, 1993).

A partir destas considerações, serão expostos os objetivos do presente trabalho.

1.1 **Objetivo**

O presente trabalho tem por objetivo principal a construção de uma ferramenta de visualização de classes (incluindo relacionamentos entre elas), métodos e atributos de um programa Java em formato tridimensional.

Compõe o escopo deste projeto o aprofundamento em assuntos relacionados à visualização de dados e, especificamente, à visualização de software.

1.2 **Motivações**

Representações visuais, sejam simples ou complexas, são importantes para a compreensão de um programa. Sua utilidade tem impacto positivo em atividades que estendem-se desde a facilitação para aprendizagem de programação até o oferecimento de meios adequados à realização de engenharia reversa em um software.

A evolução do hardware nos últimos anos possibilitou considerável avanço de qualidade no desenvolvimento de software. A visualização da informação é exemplo expoente de área

que se beneficiou com o aumento de desempenho e confiabilidade do hardware.

Com o advento de dispositivos de vídeo de alto desempenho, as imagens de alta definição, animações e outras formas sofisticadas de visualização passaram a fazer parte do cotidiano dos usuários de computador. Tal fenômeno tem despertado crescente interesse na visualização de informações que outrora poderiam ser vistas apenas em formato textual.

Neste sentido, o trabalho aqui apresentado apóia-se na popularização da visualização da informação e na imediata (quase óbvia) vantagem de se ter uma representação visual dos elementos de uma linguagem orientada a objeto.

Ainda que a possibilidade de se visualizar determinadas informações seja, por si só, motivo suficiente para o desenvolvimento desta ferramenta, há que se enfatizar na mesma medida o propósito educativo do projeto.

Sensíveis ao crescente uso das linguagens orientadas a objeto, os cursos de informática têm concentrado atenção no ensino deste paradigma OO, mesmo aos iniciantes na área. A absorção dos conceitos do paradigma depende, dentre outros aspectos, da boa capacidade de abstração do aprendiz e das técnicas empregadas por quem ensina. Com a implementação deste trabalho, tais técnicas passarão a contar com um apoio do fator visual, podendo aumentar o interesse pela matéria e facilitar seu aprendizado.

A utilização de ambiente tridimensional também constitui fator motivacional deste trabalho. A visualização de um programa em formato 3D implica na possibilidade de aproveitamento de recursos indisponíveis em ambiente bidimensional, tais como a rotação ou translação de componentes em tela. Assim, um objeto que esteja visualmente inacessível em dado momento da execução, pode ser observado através da aplicação destes recursos. A alteração da profundidade através da aplicação de zoom é outro expediente que se pode dispor para a visualização mais detalhada ou mais geral da cena na qual os objetos estão colocados.

1.3 Organização

Para contemplar sua proposta, o presente texto estrutura-se da seguinte forma: após esta introdução, a visualização de dados e seus tipos (incluindo a visualização tridimensional) são definidos e abordados no Capítulo 2. É destacada a visualização de software orientado a objeto, como argumento principal deste projeto. No mesmo contexto, a conceituação e utilização de metáforas, bem como a técnica de visualização em tridimensional,

são esmiuçadas nos aspectos relevantes ao tema apresentado.

O terceiro Capítulo sintetiza temas próprios do paradigma da orientação a objeto, formando base para a escolha da metáfora visual de representação da estrutura da linguagem Java, fundamental para a implementação deste trabalho.

Na seqüência, são descritas as etapas cumpridas para a realização deste trabalho, bem como a ferramenta desenvolvida para os propósitos do projeto. Destaca-se a descrição do funcionamento da ferramenta em seu estágio atual, inclusive pela exibição e análise do código-fonte das classes implementadas.

No quinto Capítulo é apresentada a análise dos resultados alcançados pela submissão de alguns programas Java à ferramenta, enfatizando a viabilidade de seu uso em relação a programas com diferentes perfis.

Por fim, no Capítulo 6 são feitas as considerações finais sobre o trabalho, onde seus objetivos são retomados. Além disso, são apresentadas propostas para trabalhos futuros.

2 *Visualização de dados*

É próprio do senso comum que uma boa imagem vale mais que mil palavras. Ao longo da história da humanidade, abstrações visuais têm sido desenvolvidas para auxiliarem nosso raciocínio: as pinturas na antigüidade, mapas no Egito antigo e os diagramas de Euclides são alguns exemplos (WARE, 2004).

Ao longo do tempo, muitas atividades relacionadas às abstrações visuais se desenvolveram e se tornaram disciplinas estruturadas, tais como a cartografia, o desenho mecânico e as representações de esquemas elétricos, para citar algumas.

A visualização de dados é uma destas disciplinas (WARE, 2004). A idéia de se usar um computador para mostrar dados em um formato gráfico remonta dos primeiros anos do surgimento da computação. As primeiras implementações deste conceito na prática eram determinadas pelo desenvolvimento do hardware disponível na época e tinham estreita relação com este. No começo dos anos de 1960, muitos dos conceitos básicos da computação gráfica foram esboçados numa tese de doutorado que descrevia um sistema chamado *Sketchpad* (GALLAGHER, 1994).

Por conta dos computadores, diagramas de informação ou visualizações podem ser desenhados automaticamente, no momento de seu uso. O potencial para a área é vasto e os avanços na computação gráfica têm permitido a construção de sistemas que possibilitam a visualização da informação em grande escala em áreas como a medicina, negócios, finanças, educação e outras tantas que dependem da rapidez e boa compreensão das informações para seu progresso.

Este capítulo conceitua a visualização, termos correlatos e apresenta seus tipos, além de abordar aspectos da visualização tridimensional.

2.1 Definição

Em um dicionário, é comum encontrar a definição para visualização como uma criação mental de imagem (HOUAISS; VILLAR, 2001). Na computação gráfica, o termo ganha um significado mais abrangente, não se atendo apenas aos mecanismos internos da maneira humana de pensar e compreender certo dado. De uma construção interna da mente humana, a visualização torna-se também um artefato externo, apropriado para auxiliar em tomadas de decisão (WARE, 2004). Raciocínio semelhante expressa Stuart Card no prefácio do livro *Information Visualization – Perception for design* (WARE, 2004):

A visualização da informação diz respeito à cognição externa, ou seja, como os recursos externos à mente podem ser usados para aumentar a capacidade cognitiva da mente. Conseqüentemente, o estudo da visualização da informação envolve a análise do lado da máquina e do lado humano.

Genericamente, a visualização é o processo pelo qual se representam dados – de natureza comercial ou científica – através de imagens, na intenção de melhorar a compreensão acerca desses dados. Em outras palavras, é qualquer técnica usada para a criação de imagens, diagramas e animações para comunicar uma mensagem (WIKIPEDIA, 2005c; WHATIS.COM, 2005).

Apesar da conceituação do tema não ser dispersiva e de não haver sérias discordâncias sobre seu significado, existem ainda compreensões incorretas sobre a visualização da informação, onde se destacam, segundo Fast (2004):

i) *A visualização de dados objetiva eliminar texto.* Embora esta afirmação seja comum, a finalidade é encontrar a representação apropriada para uma tarefa particular e, em algumas situações, o texto continua sendo a melhor forma de representação. Da mesma forma, não se pode assegurar que, em todos os casos, a compreensão e assimilação do exposto se dará mais rapidamente com representações gráficas do que com texto.

ii) *O uso da visualização de dados será bem justificado apenas para casos complexos, com o conjunto de dados muito extenso e que deva ser exibido de uma só vez.* Há situações em que o desafio é reduzir um grande volume de dados a um subconjunto gerenciável, para que neste seja bem aplicada a visualização.

2.2 Tipos de visualização

Um dos aspectos a ser destacado na visualização é sua larga abrangência disciplinar. Da agregação de várias ramificações do conhecimento que a compõem, nasceram tipos voltados a diferentes finalidades. Conforme Munzner (2002),

A visualização da informação tira proveito de idéias de várias tradições intelectuais, incluindo a computação gráfica, a interação entre homem e computador, a psicologia cognitiva, a semiótica, o *design* gráfico, a cartografia e a arte.

A visualização da informação ¹ e a visualização científica integram estes tipos, mas sua distinção nem sempre é elementar. A subdivisão da visualização em ramo científico e não científico não deve provocar a exclusão mútua das áreas. Em outras palavras, a visualização da informação não é uma área não-científica e a visualização científica não deixa de carregar conteúdo informativo. O que se observa é a diversidade de foco de atenção nas duas áreas. De forma sucinta, pode-se afirmar que a visualização científica foca o dado e a visualização da informação, a informação, geralmente abstrata (MUNZNER, 2002; GERSHON; EICK, 1997).

A Tabela 1 mostra indicações para a separação entre os dois tipos de visualização. Através da análise destes fatores, é possível determinar a técnica de visualização.

Tabela 1: Visualização da informação comparada com a visualização científica.

	Audiência	Tarefas	Entradas	Quantidade de entradas
Visualização Científica	Especializada e altamente técnica	Compreensão profunda do fenômeno científico	Dados físicos, medidas e simulações	Variável
Visualização da Informação	Diversa, difundida e menos técnica	Procura, descoberta, relacionamentos	Relacionamentos, dados não físicos, informação	Variável

Fonte: Gershon e Eick (1997)

Um aspecto chave nesta separação é o fato que a visualização da informação envolve facilidade de uso. Em contraste com a visualização científica - que geralmente serve cientistas altamente treinados, as interfaces criadas para manipulação da informação podem ser

¹Neste caso, o termo *visualização da informação* é usado para designar especificamente a visualização não científica.

amplamente difundidas entre uma comunidade diversificada e potencialmente não-técnica (GERSHON; EICK, 1997).

Sustentam Wiss, Carr e Jonsson (1998) que a visualização da informação, ao contrário da visualização científica, prima pela visualização de dados abstratos que podem não ter uma representação visual natural. Por conta disso, a visualização da informação requer uma atividade adicional em relação à visualização científica: a criação de um projeto de visualização da informação, que pode ser utilizado em diferentes aplicações de visualização da informação.

A visualização da informação visa a reduzir a complexidade do exame e da compreensão da informação. No conceito de Chittaro (2001), técnicas apropriadas para a exibição dos dados devem visar a alguns objetivos, tais como:

- permitir que os usuários explorem dados disponíveis em vários níveis de abstração;
- fornecer aos usuários um senso maior de comprometimento com os dados;
- fornecer aos usuários uma compreensão mais profunda dos dados;
- encorajar a descoberta de detalhes e relações que poderiam ser difíceis de se perceber sem o recurso da visualização;
- dar suporte ao reconhecimento de padrões ao explorar as capacidades de reconhecimento dos usuários.

As seções seguintes versarão sobre a visualização da informação nas áreas médica, geográfica e científica, enfatizando a utilização da visualização da informação na educação.

2.2.1 Visualização da informação na medicina

O interesse na visualização de aplicações médicas possui longa tradição e concentra-se em problemas relacionados à aquisição de imagens (por exemplo, através de tomografias) e no processamento necessário para visualizar a imagem obtida, através de técnicas como o *anti-aliasing*² e renderização (CHITTARO, 2001).

Sustenta ainda Chittaro (2001) que a visualização da informação na medicina é uma evolução ou uma extensão do campo das imagens médicas. Com sua maior abrangência,

²Técnica que visa minimizar o efeito causado por sinais contínuos diferentes que se tornam indistinguíveis quando uma imagem é exibida.

ela pode melhorar a tradicional aquisição de imagens médicas com novas capacidades, aumentando a interatividade e facilidade de uso.

Na seqüência são ilustrados exemplos de aplicações da visualização da informação na medicina.

North e Korn (1996) apresentam um exemplo de visualização interativa de imagens médicas através da interface *Visible Human Explorer*. Conforme mostrado na Figura 1, tal interface exibe uma versão miniaturizada do corpo humano em uma visão frontal em duas dimensões e, separadamente, mostra um corte transversal perpendicular ao longitudinal.

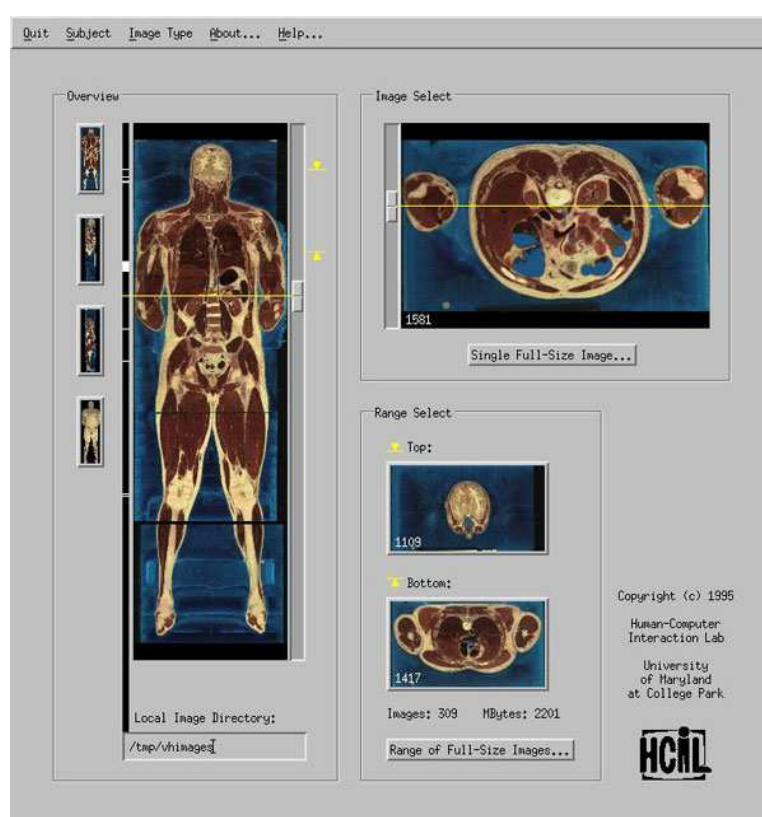


Figura 1: Exemplo de interface de visualização médica
Fonte: North e Korn (1996)

O usuário pode mover os planos longitudinal e ortogonal através de deslizadores na tela. A ferramenta também permite que se navegue por miniaturas de imagens, com a opção de recuperar as imagens correspondentes. Estes recursos permitem que o usuário explore facilmente o conteúdo de todo conjunto de dados e recupere apenas imagens do seu interesse.

Um espaço de visualização de informação para neuroimagens ³ é proposta por Nielsen

³Termo usado para as técnicas de obtenção de imagens do sistema nervoso periférico, da espinha dorsal e do cérebro.

e Hansen (1997). O ambiente não apenas permite renderizar neuroimagens tridimensionais como também auxilia o usuário na interpretação das imagens ao associá-las com informações textuais sobre elas.

A Figura 2 apresenta uma imagem gerada pela ferramenta. Através dos comandos situados no canto inferior esquerdo, o usuário pode variar a transparência e o tamanho de alguns objetos da visualização.

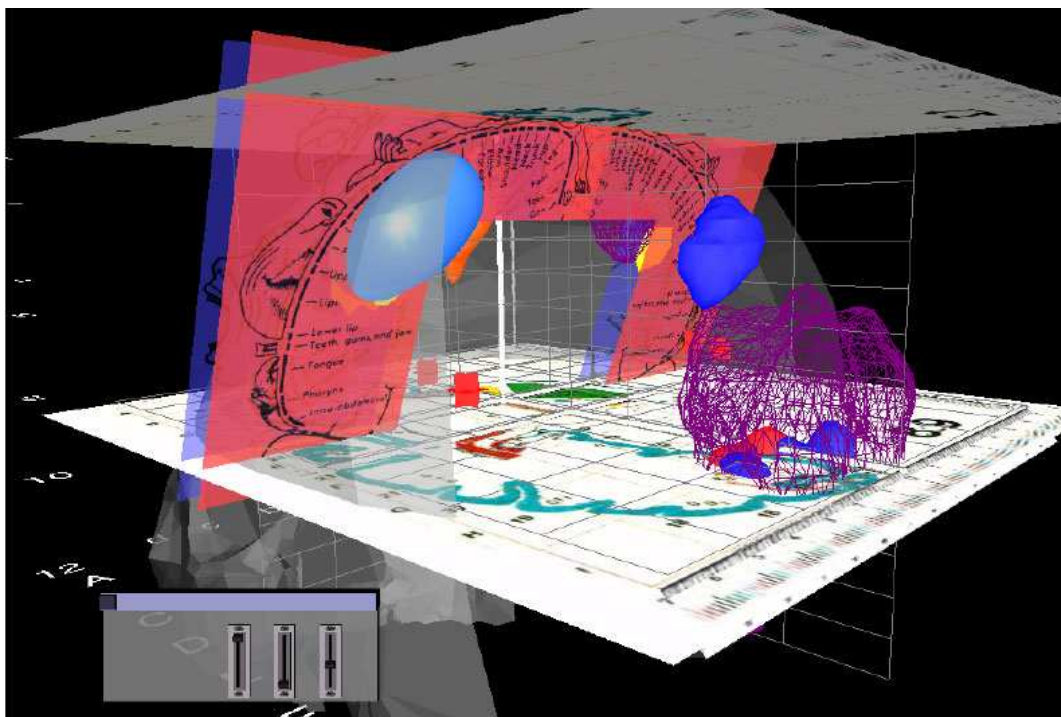


Figura 2: Visualização de neuroimagem
Fonte: Nielsen e Hansen (1997)

Destaca Chittaro (2001) que o tipo de dado relativo ao tempo (temporal) tem particular relevância na visualização da informação médica. Um dos primeiros sistemas propostos para este tipo de dado foi o *Time Line Browser*, criado por Cousins e Kahn (1991), que oferece a visualização de eventos – tal como a medida de um parâmetro clínico com seu respectivo valor – e intervalos com duração, tal como o estado clínico de um paciente, em uma linha do tempo.

Uma visualização mais elaborada do histórico do paciente é proposta no sistema *Lifelines*, de autoria de Plaisant, Mushlin e Snyder (1998). Os itens de interesse, tais como a medicação, diagnóstico, consultas e condições gerais do paciente, são organizados em áreas distintas da tela, conforme mostra a Figura 3.

É facultado ao usuário a obtenção de detalhes dos itens relacionados e a execução de *zoom* em uma determinada faixa de tempo relacionada a eventos e intervalos do histórico

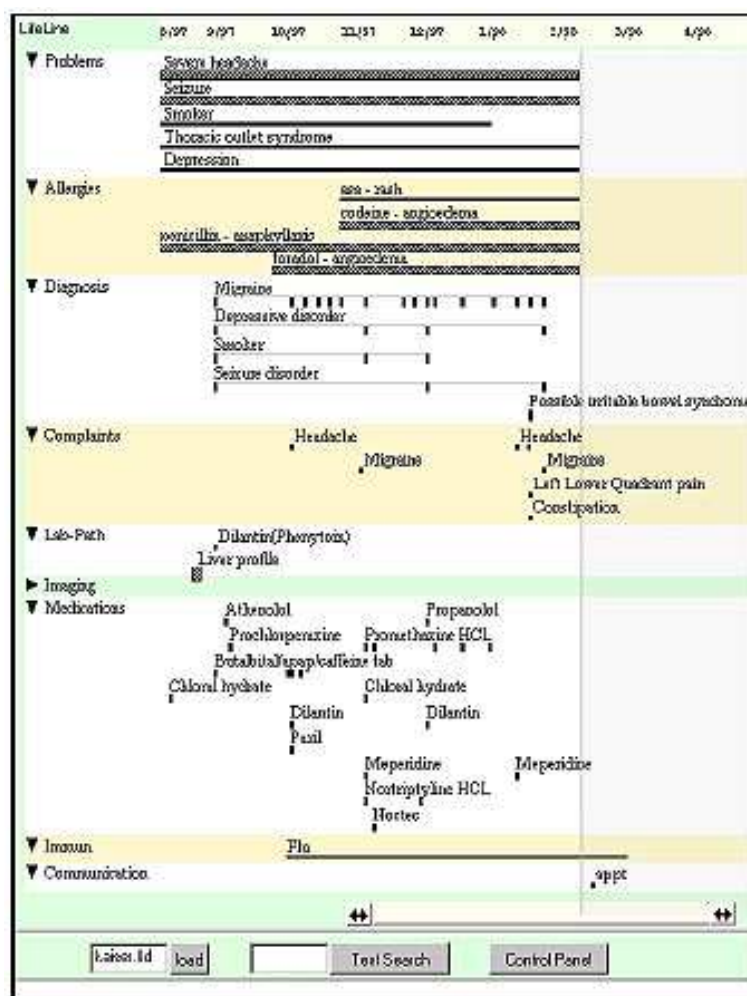


Figura 3: Sistema Lifelines
 Fonte: Plaisant, Mushlin e Snyder (1998)

do usuário.

2.2.2 Visualização geográfica

A visualização geográfica (GVis), por vezes chamada de visualização cartográfica, é uma forma de visualização de informação na qual os princípios contidos em Sistemas de Informação Geográfica (GIS), Análise Exploratória de Dados (EDA) e a própria visualização genérica de informação, são integrados no desenvolvimento e avaliação de métodos visuais que facilitam a exploração, análise, síntese e apresentação da informação relacionada à geografia (MACEACHREN; BOSCOE; HAUG, 1998).

A visualização geográfica estende as tradicionais abordagens cartográficas na representação de informações relativas à geografia ao enfatizar o uso de mapas e outras formas de representação para a construção de conhecimento e ao ligar dinamicamente a exibição

do mapa com as estruturas de dados geográficos subjacentes e com os usuários do sistema, resultando em mapas que se alteram em resposta às ações tomadas pelos usuários.

A literatura corrente apresenta diversos trabalhos direcionados ao desenvolvimento de ferramentas de análise de dados geográficos que integram princípios da cartografia, dos GIS e da EDA.

Brewer, MacEachren e Abdo (2000) propõem um ambiente de visualização geográfica colaborativo no qual grupos de cientistas à distância podem explorar informações espaço-temporais provenientes do sistema. O protótipo foi projetado para facilitar a colaboração entre usuários cujo interesse é explorar as séries de tempo de dados climáticos através de imagens animadas e interativas. Nas palavras de seus autores, “o foco do sistema é utilizar a geovisualização para mediar e aumentar a construção do conhecimento colaborativo entre cientistas ambientais” (BREWER; MACEACHREN; ABDO, 2000).

A Figura 4 apresenta um mapa onde os colaboradores podem manipular uma representação tridimensional da precipitação e temperatura enquanto variam de acordo com o terreno. Os usuários ficam aptos a controlar parâmetros de animações de séries de tempo, bem como o esquema de cores usados para representar os dados.

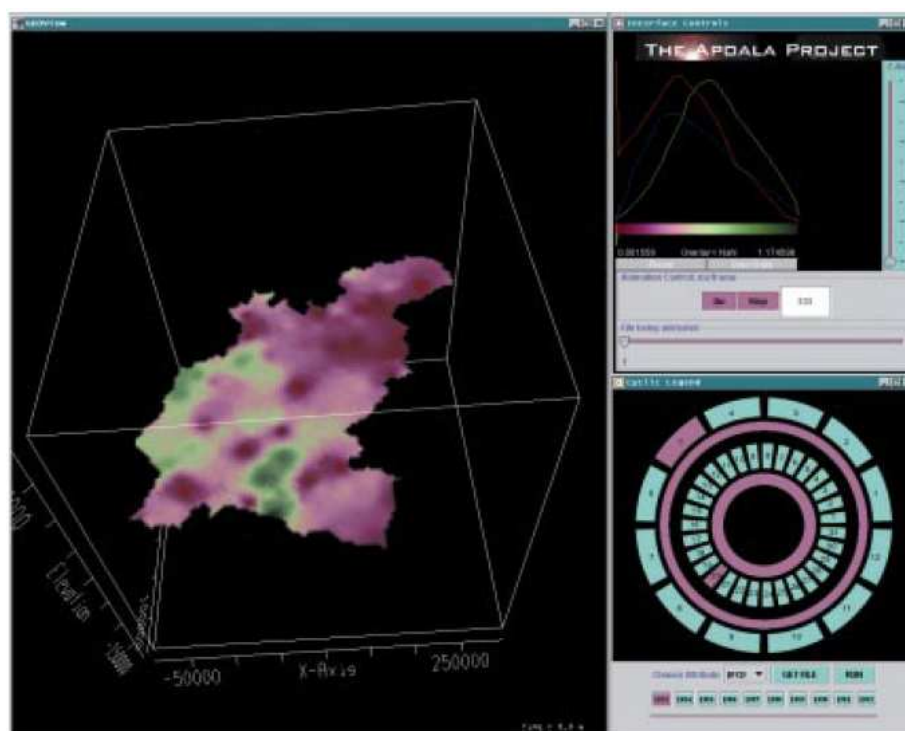


Figura 4: Visão do protótipo de ambiente colaborativo de geovisualização
Fonte: Brewer, MacEachren e Abdo (2000)

O protótipo foi construído a partir de um conjunto de ferramentas Java e Java3D,

que incluem:

- VisAD, uma biblioteca Java para visualização interativa e colaborativa de dados numéricos.
- DEMViewer, um visualizador Java de modelo de elevação digital para exportação de arquivos ASCII.
- Extensões para consulta de dados, referência a tempo e rede.

As ferramentas de consultas temporais são ligadas a uma base de dados orientada a objetos que dá suporte a uma complexa gama de consultas sobre espaço e tempo.

A fim de criar um ambiente colaborativo, Brewer, MacEachren e Abdo (2000) construíram um mecanismo para possibilitar a comunicação entre diferentes computadores. O mecanismo desenvolvido é uma aplicação Java capaz de promover a comunicação de eventos iniciados pelo usuário entre aplicações colaborativas em rede.

Como forma de avaliar o sistema, Brewer, MacEachren e Abdo (2000) entrevistaram seis especialistas na área, todos potenciais usuários da ferramenta, com o objetivo de obter suas opiniões acerca das pesquisas que poderiam ser facilitadas pelas ferramentas de geovisualização e sobre os tipos de ferramentas que seriam necessários.

2.2.3 Visualização científica

A visualização científica cuida da geração de imagens digitais representativas de fenômenos físicos e que são normalmente obtidas a partir de grandes quantidades de dados. Por sua vez, estes dados são obtidos a partir de experiências realizadas em laboratórios ou são provenientes de sensores colocados em ambientes de coletas de informações (YATES; RIBEIRO, 1999; WIKIPEDIA, 2005b).

A visualização científica teve impacto positivo na prática da ciência ao longo dos últimos anos. Ao aliar-se à força da percepção visual humana tem gerado conhecimento através da visualização de dados complexos.

Seu interesse mais imediato é suprir as necessidades da comunidade científica, mas pode ser útil em aplicações voltadas a leigos, tais como a exibição das condições climáticas e a representação da densidade populacional de certa região.

2.2.3.1 Aplicação da visualização científica na educação

Um ramo particular da atividade humana que tem sido impulsionado e auxiliado pela evolução tecnológica é a educação assistida por computador. A literatura destaca inúmeros empreendimentos voltados à finalidade educacional que utilizam a visualização científica para seu cumprimento.

Baseados no exame das necessidades de programadores experientes e considerando a trajetória pedagógica que torna um programador novato em um programador experiente, Eisenstadt, Price e Domingue (1999) desenvolveram uma abordagem de ensino baseada na visualização de informação, definida como uma coleção de técnicas que permite aos iniciantes observarem detalhadamente a execução de um programa.

Naquela pesquisa são citados diversos projetos de ITS (*Intelligent Tutoring Systems* ou Sistemas de Ensino Inteligentes) voltados ao domínio da programação de computadores que obtiveram resultados relativamente importantes combinando ciência cognitiva, inteligência artificial e interações homem-máquina.

Sustentam os autores que a abordagem típica de um ITS no ensino da programação de computadores apoiou-se em um modelo onde o estudante escreve código num editor de texto em determinada área e o ITS simplesmente imprime comentários em linguagem textual em outra área, possibilitando aos aprendizes a visualização apenas da saída do programa.

Como resultado dos esforços da pesquisa em proporcionar efetiva visualização da execução de um programa, foi desenvolvido um *framework* chamado “Viz”, no qual a execução de um programa é considerada uma série de eventos que acontecem em favor ou pela intervenção de um *player*, que pode ser uma função, uma estrutura de dados ou uma linha de código. Tais eventos são armazenados num módulo de histórico (o que permite que sejam re-executados) e mapeados numa representação visual, acessível ao usuário final.

No mesmo ano de 1999, em trabalho intitulado The SSciVEE Project (*Supportive Scientific Visualization Environments for Education*), Edelson, Pea e Clark (1999) procuraram explorar o potencial desta tecnologia para demonstrar que a visualização científica, incorporada à aprendizagem investigativa, pode tornar estudantes de diversas áreas aptos a desenvolverem uma melhor compreensão de fenômenos naturais.

Um dos desafios dessa pesquisa foi identificar as necessidades tecnológicas que permitiriam transformar a visualização científica em uma tecnologia educacional efetiva. A

transformação de ferramentas e técnicas desenvolvidas pela comunidade científica em ambientes que encorajassem os estudantes à pesquisa também foi um desafio significativo enfrentado pela pesquisa.

O objetivo colocado para o estudo foi a compreensão dos requisitos de tais ambientes de visualização científica voltados à educação. Para tanto, foram criados ambientes para o estudo de aspectos climáticos e da geografia que permitiram que os aprendizes examinassem conjuntos de dados criados pela comunidade científica.

Além disso, os estudantes puderam criar seus próprios dados através de operações aritméticas embutidas e modelos climáticos. Tal configuração, suportada por uma base de dados multimídia, permitiu que os estudantes visualizassem os dados na forma de mapas de cores.

Foram examinados os modos pelos quais a arquitetura dos ambientes de visualização causam impacto no entendimento conceitual dos estudantes e os modos como os mesmos ambientes são adotados em classe.

Na prática, a pesquisa SSciVEE gerou a ferramenta *WorldWatcher*, que tem uma janela de visualização (mostrada na Figura 5), compondo um ambiente de auxílio à visualização científica para investigação de dados científicos. Segundo Edelson, Pea e Clark (1999), a *WorldWatcher* propicia um ambiente acessível e amigável para que os estudantes explorem, criem e analisem dados científicos.

Os dados são fornecidos com a *WorldWatcher* em bibliotecas de dados que dão suporte às atividades educacionais centradas em tópicos específicos. Além disso, os usuários podem importar seus próprios dados para a *WorldWatcher*. A primeira biblioteca foi projetada para suportar investigações do clima global e mudanças climáticas.

Com o projeto, os estudantes puderam, na prática, examinar as causas e implicações das alterações climáticas. De forma mais ampla, os estudantes puderam tomar contato e familiarizarem-se com a visualização científica através de um ambiente amigável e propício ao aprendizado.

2.3 Aspectos relevantes no projeto de uma aplicação de visualização

Segundo Maletic, Leigh e Marcus (2001a), quando se objetiva a construção de uma ferramenta de visualização que atenda às necessidades do usuário, o projetista deve con-

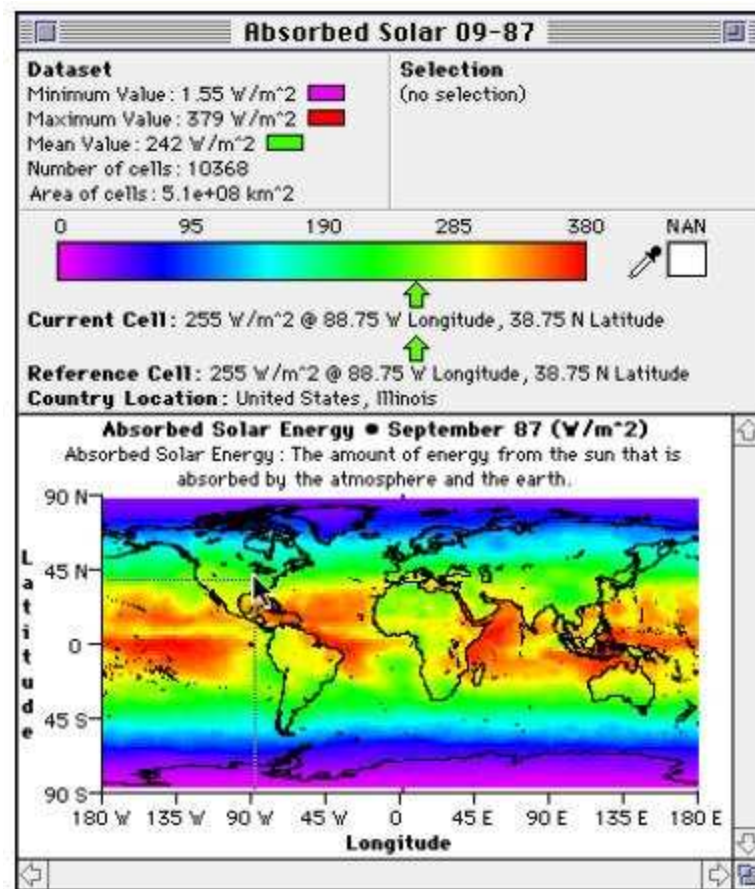


Figura 5: Uma janela de visualização da ferramenta *WorldWatcher*
 Fonte: Edelson, Pea e Clark (1999)

siderar os seguintes aspectos:

- **Visão geral:** deve ser oferecida uma visão geral da coleção de dados a ser representado.
- **Zoom:** é necessário que se permita a visualização em detalhes de determinados itens de interesse do contexto da visualização. Quando se aplica o zoom, é importante que o contexto global possa ser recuperado através, por exemplo, de uma função que retorne a visualização para global – e não mais detalhada. Para tanto, as metáforas devem ser projetadas de tal forma a não perderem o significado quando forem aumentadas ou diminuídas em sua visualização.
- **Filtro:** a intenção de se aplicar filtros é a de separar itens de menor interesse da visualização. A filtragem feita pela remoção de partes da visualização irá causar distúrbios no contexto global. Portanto, é importante a verificação prévia da capacidade do sistema em suportar algum tipo de abstração das partes removidas.

- **Detalhes sob demanda:** significa dar ao usuário a possibilidade de selecionar um item ou grupo e obter detalhes destes quando necessário.
- **Relações:** trata-se da visão dos relacionamentos entre itens. Para uma estrutura hierárquica de dados, é necessário que a visualização mostre os relacionamentos do tipo pai-filho.
- **Histórico:** reflete a manutenção de um histórico de ações para que seja possível desfazer uma ação.
- **Extração:** permite a extração de sub-coleções e de parâmetros de consulta. Esta tarefa preocupa-se com o estado atual da visualização e é relacionada apenas com a aplicação. A forma com que os dados são visualizados não afetam o estado da visualização.

Além destes aspectos, considerados como tarefas na elaboração de uma ferramenta de visualização, o tipo de dado deve também ser considerado. Segundo Chittaro (2001), sete tipos de dados são identificados nos itens a serem exibidos:

- uma dimensão (*1-dimensional*): dados lineares organizados de uma maneira seqüencial, tais como uma lista alfabética de nomes, códigos-fonte ou documentos textuais.
- duas dimensões (*2-dimensional*): mapa de dados abrangendo parte de uma área, tais como mapas e fotografia.
- três dimensões (*3-dimensional*): dados com volume e com relações potencialmente complexas uns com os outros, tais como as moléculas, o corpo humano ou construções.
- temporal: dados com um momento de início, momento de fim e possíveis sobreposições em uma escala de tempo, tais como aqueles encontrados em registros médicos, gerenciamento de projeto ou edição de vídeo.
- multidimensional: dados com n atributos que se tornam pontos em um espaço n -dimensional, tais como registros em bases de dados relacionais.
- árvore: coleções de itens ligados hierarquicamente por uma estrutura de árvore, tais como diretórios, organogramas e árvores genealógicas.
- rede: coleção de itens ligados por uma estrutura de grafo, tais como redes de telecomunicações, a rede mundial de computadores ou estruturas de hipermídia.

Ressalta Geisler (1998) que, provavelmente, a forma mais comum de dado unidimensional é um documento de texto. Na maioria dos casos, não há necessidade de se usar visualização para documentos de texto, já que são simplesmente lidos do início ao fim, ou partes específicas são referenciadas quando necessário. No entanto, em casos específicos, é possível usar a visualização para oferecer ao usuário a possibilidade de ver aspectos ou fazer correções em partes do documento que, de outra forma, seria difícil.

Para ilustrar tal consideração, Geisler (1998) cita o projeto Shakespeare Virtual, desenvolvido pelo Grupo de Linguagens Visíveis (*Visible Languages Group*), do *Massachusetts Institute of Technology* (MIT). Baseado em métodos que permitem a exibição de tipografia em qualquer tamanho, posição e orientação, o projeto captura as peças completas de William Shakespeare – cuja quantidade de texto é da ordem de milhões palavras – e fornece ao usuário a possibilidade de visualização de seus padrões e de suas estruturas individualmente.

De acordo com Chittaro (2001), os itens de dados em cada categoria podem possuir múltiplos atributos. Um objeto tridimensional pode ter atributos adicionais tais como cor, nível de transparência e brilho. Da mesma forma, um nó em uma árvore também pode ter outros atributos, incluindo nome, data de criação e modificação, entre outros.

Assim, a separação entre categorias diferentes não é sempre estrita. Por exemplo, dados temporais podem ser também vistos como uma instância de dados multidimensionais. No entanto, esta separação é útil para orientar a escolha das técnicas de visualização de informação, ou seja, quando o aspecto temporal é dominante no conjunto de dados considerado, técnicas de exibição que dão um papel central ao tempo podem fornecer melhores resultados do que técnicas mais genéricas que não assumem relações específicas entre múltiplos atributos.

2.4 Visualização de software

A expressão *visualização de software* pode assumir várias conotações, dependendo de quem a interpreta. Maletic, Leigh e Marcus (2001a) descrevem o termo como a exibição gráfica de informações sobre o sistema de software, que incluem sua estrutura, sua execução e até mesmo seu próprio código-fonte.

Diversos outros autores conceituam o termo e destacam sua importância na computação moderna.

Por muitos anos, o principal meio de se tentar entender um programa de computador ou execução de algum processo era pelo exame de seu código-fonte e pela utilização de um depurador. Recentemente, um bom número de sistemas de visualização e animação de processos computacionais foram desenvolvidos para auxiliar na compreensão de um programa (STASKO, 1993).

Este excerto de Stasko (1993) revela o surgimento de um novo conceito em visualização da informação, que ajudaria na compreensão do funcionamento de um software.

Na mesma linha de raciocínio, porém incluindo novo aspecto e dando outros termos à definição, Stasko (1993) sustenta que a visualização de software ilustra dados e processos de computador. A visualização de programas ilustra as estruturas de dados, o estado do programa e também seu código-fonte.

Referindo-se à aplicabilidade da visualização na compreensão de programas, a doutrina de Nascimento e Ferreira (2004) prescreve que

a visualização de software, como ferramenta da engenharia de software, auxilia o programador a analisar e a entender a estrutura e o funcionamento de um programa, em um nível maior de abstração do que quando comparada a uma simples leitura do código-fonte.

Expressando-se de forma mais genérica, Price, Baecker e Small (1998) apresentam a seguinte definição de visualização de software:

A visualização de software é o uso conjunto de características da tipografia, *design* gráfico, animação e cinematografia com a moderna tecnologia de interação homem-computador com o objetivo de facilitar a compreensão humana e o efetivo uso do software de computador.

Balzer, Noack e Deussen (2004) conceituam e destacam as várias abordagens da visualização de software:

A visualização de software, enquanto sub-área da visualização da informação, é o uso de várias formas de imagens com o objetivo de facilitar a compreensão de sistemas de software. Existem várias abordagens de visualização mostrando diferentes aspectos dos programas, tais como sua estrutura estática, o comportamento na execução, a evolução ou o processo de desenvolvimento.

Em termos gerais, a visualização da informação é um mapeamento do dado para uma forma visual que o ser humano pode observar. A Figura 6 descreve tais mapeamentos e serve como um modelo de referência para a visualização.

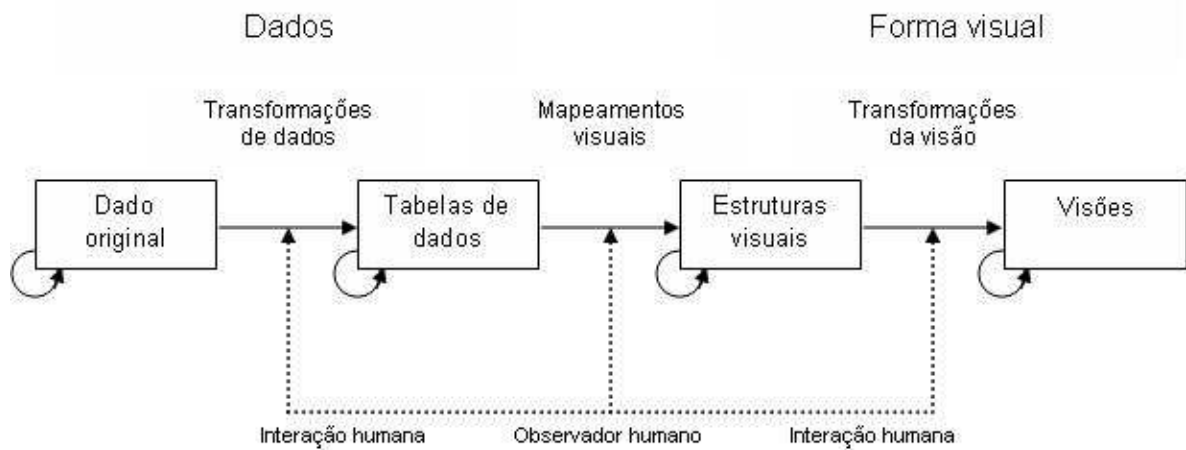


Figura 6: Modelo de referência para visualização
 Fonte: Maletic, Marcus e Collard (2002)

A primeira transformação converte os dados originais em tabelas de dados mais úteis. O dado original encontra-se normalmente em algum formato específico do domínio da aplicação, geralmente difícil ou impossível de se trabalhar.

As tabelas de dados são representações relacionais destes dados originais. As informações acerca das características dos dados (metadados) podem ser incluídas nestas tabelas de dados. A partir daí, os mapeamentos visuais transformam as tabelas de dados em estruturas visuais, ou seja, elementos gráficos.

Finalmente, as transformações de visão criam visões das estruturas pela especificação de parâmetros como posição, rotação, escala, etc.

A visualização de software mapeia diretamente este modelo de referência. Os dados originais referem-se ao código-fonte do software. As tabelas de dados, que são uma abstração dos dados originais, tomam a forma de árvores sintáticas abstratas, grafos de dependência de programas ou relacionamentos entre classes, por exemplo. As estruturas visuais são as visualizações de software já renderizadas (MALETIC; MARCUS; COLLARD, 2002).

A definição de visualização remete a diversos tópicos, tais como visualização de dados, visualização de código e programação visual, entre outros. Em razão de sua variedade de aplicações e de sua abrangência multidisciplinar, foram especificadas várias dimensões para a visualização de software.

De acordo com Price, Baecker e Small (1998), as dimensões são as seguintes: tarefas, audiência, alvo, representação e meio.

- **Tarefas** – esta dimensão define porque a visualização é necessária. Em outras palavras, ela especifica quais tarefas particulares da Engenharia de Software são suportadas pelo sistema de visualização de software. Em geral, cada sistema de visualização dá suporte ao entendimento de um ou mais aspectos de um software.
- **Audiência** – com base na tarefa suportada, a ferramenta de visualização de software pode ser ajustada para diferentes tipos de usuários. A dimensão da audiência define os atributos dos usuários do sistema de visualização. Se a tarefa suportada é a educação, os estudantes e/ou instrutores compõem a audiência. No ambiente industrial, a audiência será de desenvolvedores, pessoal de teste ou gerentes de equipe. Além disto, ferramentas diferentes podem ser designadas a usuários com diferentes níveis de habilidade.
- **Alvo** – o alvo de um sistema de visualização de software define quais aspectos do software são visualizados. O alvo é um produto de trabalho, um artefato, ou parte de um ambiente de sistema. Corresponde à fonte dos dados (ao dado original no modelo de referência apresentado). Nesta dimensão, consideram-se alvos da visualização a arquitetura, o projeto, o algoritmo, o código-fonte, os dados, as medidas e métricas. Os sistemas mais simples objetivam representar o código-fonte em um formato mais simples de entender e legível para o usuário.
- **Representação** – dependendo dos objetivos do sistema de visualização de software, o tipo de usuário e do meio disponível, um formato de representação precisa ser definido para melhor transmitir a informação ao usuário. Esta dimensão define como a visualização é construída, levando-se em conta a informação disponível. A representação manifesta-se como as estruturas visuais no modelo de referência. Uma abordagem mais detalhada deste assunto é fornecida na seção 2.6 deste trabalho.
- **Meio** – o meio é onde a visualização é renderizada. Os meios geralmente usados pelos sistemas de visualização de software são: papel, monitores monocromáticos, monitores coloridos, múltiplos monitores e *displays* de alta resolução, como telas de plasma e projetores. Outros meios atualmente investigados para uso em sistemas de visualização de software são os *displays* estéreo e ambientes imersivos de realidade virtual. Cada um destes meios possuem características diferentes e, em consequência, são apropriados para diferentes tarefas. Por exemplo, o papel e monitores mais modestos são indicados para representações estáticas em pequena escala e dimensão, enquanto ambientes virtuais imersivos oferecem condições de visualização

de estruturas maiores (como, por exemplo, grafos conectados) e meios de se fazer uso de outras entradas sensoriais, tais como o som e cheiro.

Panas, Berrigan e Grundy (2003), utilizando o termo *visualização de programa*, afirmam que, nesta atividade, a escala e a complexidade do software a ser visualizado são questões prementes, assim como o problema de sobrecarga de informações que tais questões podem trazer à tona. Salientam os autores que o uso de metáforas é fator importante na abordagem da complexidade do software.

O grande desafio de se proporcionar boa visualização de softwares complexos reside na representação compreensível e sem distorção de grande quantidade de dados. Isto implica, segundo Balzer, Noack e Deussen (2004), em se atingir duas metas: (i) a densidade de informação nas visões deve ser maximizada, considerando-se, no entanto, a restrição da compreensibilidade. Porém, até mesmo com uma grande densidade de informação, nem todos os detalhes dos dados em uma visão podem ser representados e (ii) fornecer navegação intuitiva, que permita ao usuário mover-se facilmente entre visões em diferentes níveis de abstrações e em diferentes partes do sistema visualizado.

Complementam Balzer, Noack e Deussen (2004) que existe, se não uma exclusão mútua, ao menos um intercâmbio entre a densidade da informação e a compreensibilidade. Algumas visualizações tridimensionais possuem uma alta densidade de informação, mas parecem atravancadas, ocultam informações e não fornecem visão global. Por outro lado, várias visualizações com duas dimensões são bastante claras, mas revelam poucas informações.

2.4.1 Visualização de software orientado a objeto

A representação gráfica de elementos de um software torna-se uma tarefa mais natural quando se trata de um sistema orientado a objetos. Existe uma correspondência natural entre os objetos do software e os objetos visuais que os representam (WARE; HUI; FRANCK, 1993).

Diversos projetos de visualização de software orientado a objeto foram levados a cabo e, desde então, têm sido aprimorados no mesmo passo do desenvolvimento das formas de visualização.









Ware, Hui e Franck (1993) realizaram pesquisas no sentido de desenvolver um protótipo para a visualização e manipulação no espaço tridimensional de grafos com múltiplos atrib-

utos nos nós e arcos. Tais grafos eram usados para representar módulos e relações de herança entre classes.

Mais recentemente, Maletic, Leigh e Marcus (2001a) construíram uma linguagem de representação visual chamada COOL (*Language for Comprehending OO Software*) que define um mapeamento formal para uma linguagem orientada a objeto, tal como C++ ou Java, para visualização em realidade virtual.

O sistema usa plataformas em tamanhos proporcionais ao tamanho da classe. Atributos de uma classe são representadas por esferas e as colunas representam funções-membros. A Tabela 2 mostra as entidades existentes no sistema COOL.

Tabela 2: Descrição das entidades no sistema COOL

Nome	Visualização	Significado
Plataforma		Classe
Tamanho da plataforma		Número de métodos mais número de atributos
Esfera		Atributo
Tamanho da Esfera		Tipo de atributo
Coluna branca		Construtor de função-membro
Coluna verde		Acesso a função-membro
Coluna púrpura		Modificador de função-membro
Tamanho da coluna		Linhas lógicas de código por método
Localização da esfera/coluna		Ocultação da informação

Fonte: Maletic, Leigh e Marcus (2001a)

Os diferentes tipos de componentes são codificados por cores: branco para constru-

tores, verde para controladores de acesso e púrpura para modificadores.

As relações entre classes, sobrecarga de métodos e outros aspectos próprios do paradigma de orientação a objeto são visualizados através de metáforas.

O sistema COOL possui uma linguagem de visualização com muitas camadas de abstração. A primeira é baseada na idéia de um diagrama de classes. A segunda camada de abstração é baseada em métricas do tamanho e de linhas de código por função.

2.5 Visualização tridimensional

Avanços tecnológicos na computação gráfica tornaram a visualização da informação em três dimensões viável para computadores pessoais (WISS; CARR; JONSSON, 1998).

Tal realidade permitiu que um público com pretensões menos técnicas tomasse contato com a área. A possibilidade de se visualizar em três dimensões um fenômeno natural, um gráfico de desempenho ou mesmo uma animação voltada ao entretenimento tem atraído a atenção para este ramo promissor da atividade computacional.

Nas seções seguintes serão descritos conceitos próprios do espaço tridimensional, com foco permanente nos aspectos pertinentes à visualização de dados.

A visualização e os métodos orientados a gráfico são comuns na engenharia de software. Na maioria dos casos, estes métodos usam a tecnologia bidimensional (2D), cuja grande vantagem é necessitar apenas de papel e lápis para ser efetivada.

Embora programas de projeto auxiliado por computador (CAD) utilizem a tecnologia tridimensional de visualização, a mesma é raramente usada na visualização de grandes sistemas computacionais. Em contraste com os produtos construídos pela engenharia clássica, os produtos de software são abstratos e não possuem um corpo físico. Já que não existe uma aparência natural em três dimensões, torna-se difícil encontrar conceitos visuais satisfatórios (ALFERT; ENGELEN, 2001).

2.5.1 Fundamentos de ambientes 3D

O espaço tridimensional é formado pelos eixos x , y e z , conforme mostrado na Figura 7. O ponto onde os eixos se cruzam chama-se origem e suas coordenadas são 0,0,0.

Os dois eixos usuais para visualização em duas dimensões são x e y , sendo x horizontal, com unidades positivas voltadas para a direita e unidades negativas à esquerda da origem.

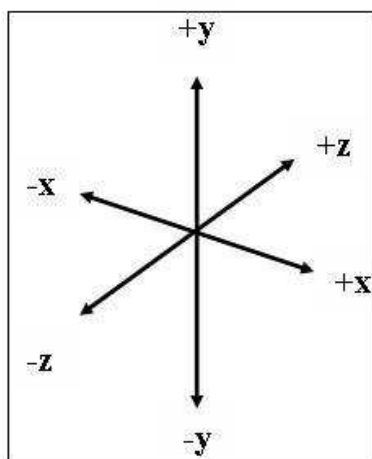


Figura 7: Os eixos do espaço tridimensional
Fonte: Balint (2001)

O eixo y é vertical, com os números positivos acima da origem e os negativos abaixo dela. O eixo z é o que caracteriza a terceira dimensão. Na visualização tridimensional, os valores negativos são os que estão na direção do observador e os positivos na direção oposta (BALINT, 2001).

Esta seção abordará aspectos fundamentais do aspecto tridimensional de visualização de objetos, sem ter a intenção de aprofundar-se em discussões de caráter essencialmente técnico, tais como tópicos da matemática e da geometria avançadas. A real intenção é destacar especificidades do espaço 3D não disponíveis no espaço de duas dimensões. Segundo Alfert e Engelen (2001), tais aspectos específicos são encontrados na(o):

- transparência dos objetos, o que revela a percepção do que está dentro, contrastando fortemente com com objetos coloridos individualmente em 2D, como mostram as Figuras 8 e 9.

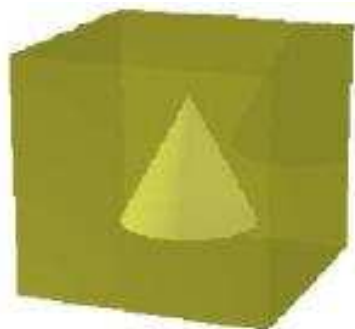


Figura 8: Transparência no espaço 3D
Fonte: Alfert e Engelen (2001)

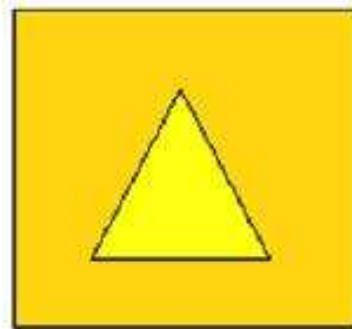


Figura 9: Transparência no espaço 2D
Fonte: Alfert e Engelen (2001)

- efeito de profundidade, cujo uso mais se dá no ambiente 3D, conforme se observa nas Figuras 10 e 11.

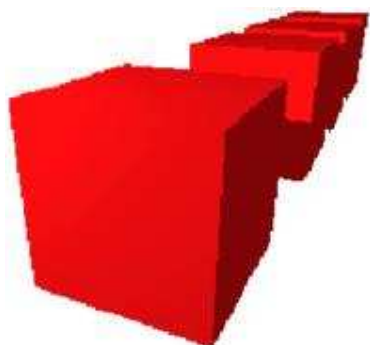


Figura 10: Efeito de profundidade no espaço 3D
Fonte: Alfert e Engelen (2001)

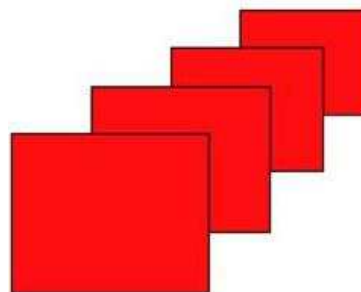


Figura 11: Efeito de profundidade no espaço 2D
Fonte: Alfert e Engelen (2001)

- movimento, importante para realização das mudanças de ponto de vista do usuário em relação ao objeto.

2.6 Metáforas

Em sentido lingüístico, o termo metáfora remete a uma comparação entre dois sujeitos, objetos ou assuntos aparentemente sem conexão. Normalmente, o primeiro objeto é descrito como sendo o segundo objeto (WIKIPEDIA, 2005a). Num dicionário de sinônimos, metáfora é definida como a transferência de sentido de um termo para outro, numa comparação implícita (HOUAISS; VILLAR, 2001).

Nos domínios da computação, o mapeamento de um modelo de programa (nível mais baixo de abstração) para uma imagem (nível mais alto de abstração) é definido por uma metáfora. Uma metáfora bem concebida é fator altamente relevante em sistema de visualização de informação. Destacam Panas, Berrigan e Grundy (2003) que o sucesso e qualidade de qualquer visualização depende da animação, das metáforas, da interconexão, interação e escala dinâmica. No entanto, os fatores mais vitais para tornar programas de visualização bem sucedidos é a recuperação dos dados necessários para visualização e a disponibilidade de uma metáfora adequada.

É essencial para todas as visualizações aqui tratadas que as características visuais das imagens sejam retrato da informação visualizada. O uso de metáforas provenientes do mundo real torna este processo particularmente intuitivo e efetivo, pois permite ao visualizador transferir habilidades de percepção pré-adquiridas para a compreensão da

visualização (BALZER; NOACK; DEUSSEN, 2004).

Ainda que boa parte das técnicas e ferramentas de visualização seja baseada na metáfora de grafo (PANAS; BERRIGAN; GRUNDY, 2003), a literatura destaca metáforas mais complexas e realistas, tais como notações de cidade em três dimensões, sistema solar, video games, caixas aninhadas, espaço 3D, entre outros. A primeira representação será abordada na próxima seção.

2.6.1 Exemplos de metáforas

As metáforas, quando representam mundos reais e estabelecem interação social, especialmente em realidade virtual, tornam-se muito importantes (KNIGHT; MUNRO, 2000). No entanto, a escolha da metáfora é essencial para aumentar a usabilidade de um sistema. Um problema fundamental de muitas representações é o fato de não terem uma interpretação intuitiva, obrigando o usuário a estar treinado para entendê-las.

Metáforas encontradas na natureza ou no mundo real evitam tal dificuldade. Panas, Berrigan e Grundy (2003) criaram a metáfora mostrada na Figura 12. Trata-se de uma cidade em ambiente tridimensional onde os prédios denotam componentes (principalmente classes Java) e a própria cidade representa um pacote (package). Diferentes metáforas entre o código-fonte e a visualização são possíveis, ou seja, os componentes não devem sempre ser mapeados para prédios e pacotes para cidades. É possível, por exemplo, mapear prédios para representarem métodos.



Figura 12: Metáfora de cidade 3D
Fonte: Panas, Berrigan e Grundy (2003)

Para dar suporte ao usuário através de uma interpretação intuitiva do software e para incrementar o realismo da metáfora, foram adicionadas árvores, ruas e postes de

iluminação ao ambiente.

Além disto, a Figura 12 ilustra informação estática e dinâmica do programa. De um ponto de vista estático, o tamanho dos prédios dá a idéia da quantidade de linhas de código de diferentes componentes. A densidade dos prédios em certa área mostra o acoplamento entre componentes, sendo que esta informação pode facilmente ser recuperada por análises métricas.

A qualidade da implementação dos sistemas junto aos vários componentes é visualizada através das estruturas dos prédios, isto é, prédios velhos e em ruínas indicam que o código-fonte do componente deve ser reconstruído.

Do ponto de vista dinâmico, carros se movendo pela cidade indicam uma execução do programa. Carros originários de diferentes componentes deixam rastro em diferentes cores. Desta forma, sua origem e destino podem ser facilmente determinados. Um tráfego denso indica comunicação intensa entre vários componentes. O desempenho e a prioridade são representados através da velocidade e o tipo dos veículos. Ocasionalmente exceções ocorrem, quando os carros colidem com outros carros ou com os prédios, provocando explosões.

A cidade vista de um satélite pode ser observada na Figura 13, onde as cidades (pacotes) são conectadas por ruas (representando chamadas bidirecionais) ou cursos de água (chamadas unidirecionais). A informação entre cidades é passada via barcos e veículos. Novamente, a informação dinâmica, bem como a estática, são ilustradas. Nuvens na figura cobrem as cidades que não são de interesse atual ao usuário, que são conseqüentemente escondidas.

O usuário deve ter liberdade total em aumentar e diminuir o zoom e navegar pelo sistema, até mesmo estar apto a usar o sistema não apenas no monitor comum, mas também num ambiente virtual.

Exemplo relevante de metáfora visual de imediata compreensão é proposto por Balzer, Noack e Deussen (2004). Focalizando um dos aspectos da visualização de software citados na seção 2.4, os autores propõem uma técnica de visualização tridimensional da estrutura estática de programas orientados a objeto, usando distribuições de objetos tridimensionais em um plano.

Tal plano, como desejam os autores, é a representação da superfície terrestre, denominada paisagem (*landscape*) no referido trabalho. Nele, a característica da metáfora da paisagem melhor explorada na visualização de um software foi a hierarquia inerente



Figura 13: A cidade tridimensional na visão aérea
Fonte: Panas, Berrigan e Grundy (2003)

aos continentes, países, estados, cidades e assim por diante, até um nível de abstração pequeno o bastante para que uma casa seja representada.

Sustentam Balzer, Noack e Deussen (2004) que a estrutura hierárquica de um software orientado a objeto é naturalmente mapeada em termos de paisagens, podendo assim serem representadas de forma compreensível e com uma distorção mínima. Argumentam ainda que, sendo os seres humanos familiares com os diferentes níveis de abstração contidos no plano terrestre, a navegação entre eles torna-se facilitada.

A Figura 14 mostra a estrutura de um software orientado a objetos através da metáfora da paisagem. Os *layouts* são baseados na hierarquia dos pacotes (*packages*), classes, métodos e atributos do software visualizado. Este software em questão possui 52 pacotes, 546 classes, 4856 métodos e 2588 atributos, podendo ser classificado como um sistema de grande porte.

A hierarquia dos pacotes é representada por esferas aninhadas. A esfera mais externa representa a raiz desta hierarquia. Esta esfera mais externa contém outras esferas que estão contidas no pacote raiz. As esferas dos pacotes do segundo nível contém esferas do terceiro nível, e assim por diante. Esta visualização aninhada é possibilitada pelo uso do recurso de transparência.

O último exemplo de utilização de metáforas em ferramentas de visualização remete a Malloy e Power (2005), que propuseram uma estratégia para a visualização de relações dinâmicas de objetos em programas Java. A metáfora de uma molécula é usada para auxiliar na compreensão destes programas. A estratégia foi implementada através da

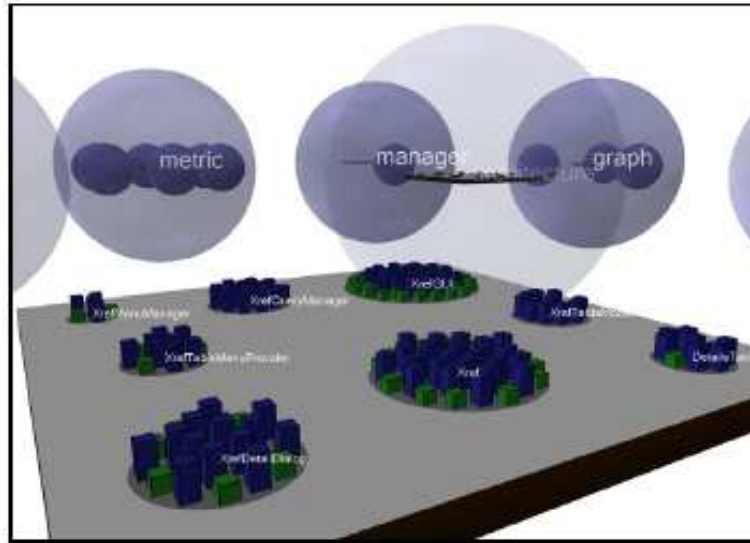


Figura 14: Vista aproximada da estrutura de um software orientado a objetos
Fonte: Balzer, Noack e Deussen (2004)

instrumentação do *bytecode* Java. A partir de então, programa é visualizado usando-se recursos tridimensionais. A Figura 15 mostra a visualização tridimensional do diagrama de objetos de um programa escolhido pelos autores como exemplo.

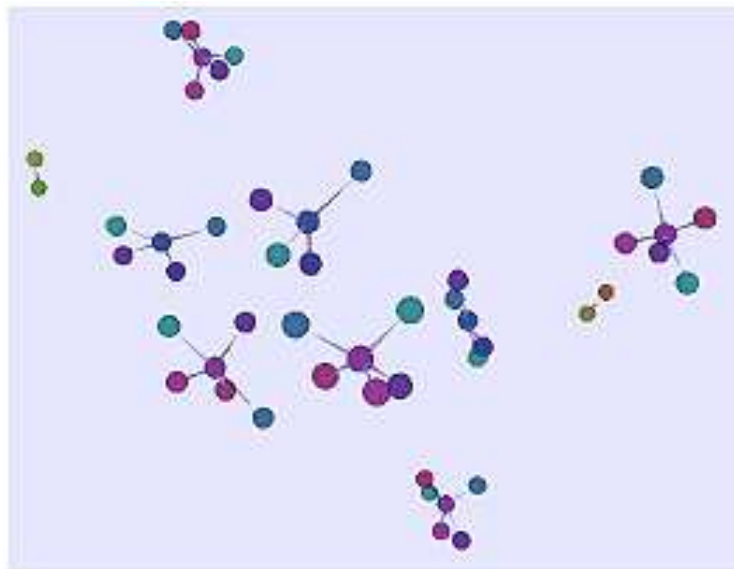


Figura 15: Diagrama de objetos de um programa em visualização tridimensional
Fonte: Malloy e Power (2005)

As analogias utilizadas na metáfora são bem diretas. Classes correspondem aos elementos químicos, os objetos correspondem aos átomos e as ligações entre objetos - baseadas em atribuições nos campos - correspondem às ligações químicas entre átomos.

O presente capítulo tratou, de forma geral, da visualização da informação e seus tipos.

Foram examinados trabalhos de autores que tratam da visualização de software e apresentados diversos exemplos de metáforas representativas de componentes de programas. O capítulo seguinte abordará o paradigma de orientação a objeto, situando-o como uma das bases deste projeto.

3 *Programação orientada a objeto*

A Programação Orientada a Objeto (POO) é um paradigma ¹ de programação de computadores na qual o software é modelado como um conjunto de objetos que interagem entre si. Tais objetos são criados a partir de modelos, que, de forma simplificada, representam objetos, pessoas ou itens usualmente presentes em nosso cotidiano (SANTOS, 2003).

O paradigma da orientação a objeto foi implantado pela primeira vez na Simula 67, uma linguagem projetada para fazer simulações. Anos depois, outras linguagens já existentes – incluindo ADA e Lisp – começaram a implantar características de orientação a objeto.

Mais recentemente, algumas linguagens foram criadas como sendo orientadas a objeto “puras”, ainda que mantendo compatibilidade com a metodologia procedimental. Algumas dessas linguagens são a Python e a Ruby. Além da linguagem Java, as mais recentes criadas sob o paradigma de orientação a objeto são o Visual Basic .NET e o C#, projetadas para a plataforma .NET, da Microsoft.

Segundo Wilson (2000), os benefícios da programação orientada a objeto incluem:

- capacidade e facilidade de reutilizar porções de código em diferentes programas, economizando tempo na análise, projeto e desenvolvimento de sistemas;
- capacidade de se dividir grandes projetos entre equipes de desenvolvedores;
- capacidade de criar meios simples e consistentes de interfaceamento entre diferentes tipos de objetos; e
- construção de software de alta qualidade.

¹A POO é freqüentemente chamada de paradigma ao invés de estilo ou tipo de programação para enfatizar o fato de que a POO altera efetivamente o modo que o software é desenvolvido, em comparação ao paradigma procedimental.

Na seqüência, serão apresentados conceitos sobre o paradigma, incluindo descrições sobre os componentes de um programa orientado a objeto.

3.1 Conceitos

3.1.1 Objetos

A orientação a objeto é um meio natural de se imaginar o mundo e de se escrever programas (DEITEL; DEITEL, 2002). Nos termos da programação orientada a objeto, objeto é uma instância de uma classe (MICROSOFT, 2001).

Tomando um carro como objeto do mundo real, a palavra que o define pode referir-se a um conceito geral de carro, como uma *classe* que não se refere a nenhum tipo específico. Outras vezes, o termo refere-se a um tipo específico de carro. No âmbito da orientação a objeto, o termo objeto ou instância refere-se a um carro particular. Assim, as características da identidade, comportamento e estado compõem um meio útil de se pensar em termos de objetos.

A identidade é a característica que distingue um objeto de todos os outros da mesma classe. Por exemplo, dois carros podem ser do mesmo modelo, marca, cor e ano de fabricação, mas terão um registro único e diferente entre ambos.

O comportamento é uma característica que torna o objeto útil. Objetos existem a fim de fornecer um comportamento e tal comportamento é, por assim dizer, a parte acessível do objeto. O comportamento de um objeto também pode determinar sua classificação e objetos da mesma classe compartilham da mesma classificação.

Por fim, o estado refere-se às funcionalidades internas de um objeto que o habilitam a fornecer seu comportamento definido. Um objeto bem projetado mantém seu estado inacessível e tal fenômeno está intimamente ligado aos conceitos de abstração e encapsulamento, abordados nas seções 3.1.2 e 3.1.3, respectivamente. Dois objetos podem, coincidentemente, conter o mesmo estado mas, não obstante, serem dois objetos diferentes. Exemplificando, dois gêmeos idênticos contém exatamente o mesmo estado, mas são duas pessoas distintas (MICROSOFT, 2001).

3.1.2 Abstração

Sustenta Wilson (2000) que abstração é o processo de desenvolvimento de uma representação simples daquilo que é complicado. Um exemplo disto é a arte abstrata. Não é a intenção do artista abstrato descrever uma pessoa ou cena como uma fotografia. A real intenção é representar aqueles detalhes ou atributos que são importantes para o artista.

Seguindo o mesmo raciocínio, no universo da programação deve-se considerar quais atributos de um objeto são importantes para a tarefa em questão. E quais atributos não são. A partir daí, deve-se representar os atributos críticos do objeto através de código de programa.

Um programador desenvolve uma abstração determinando quais atributos de um objeto são significativos. Isto significa que não há um único caminho certo de se abstrair um objeto.

A qualidade de uma abstração é muito importante para o projeto de software. Se o programa necessita de meios de distinguir a diferença entre um débito e um crédito e for criada uma abstração que não faça esta distinção, então a abstração não servirá aos propósitos do programa. Da mesma forma, se a abstração faz distinções que jamais são usadas (tal como se ter objetos separados para empregados com cabelos loiros e cabelos morenos), então a abstração é provavelmente mais complicada do que deveria ser. Ao se projetar qualquer sistema, o mais simples é geralmente o melhor e menos propenso a erros.

O processo de desenvolvimento de uma abstração envolve a procura de objetos passíveis de representação, a determinação de quão diferentes e quão semelhantes eles são. Além disso, deve-se considerar como os objetos são relacionados. Um objeto é de fato um tipo de outro objeto, assim como uma raposa é um tipo de mamífero? Assim, o programador deve classificar os objetos que ele deve representar.

Por conta da complexidade de muitos sistemas, é pouco comum a criação de uma abstração perfeita na primeira tentativa. O desenvolvimento de uma abstração é, normalmente, uma questão de refinamentos e aproximações sucessivas (WILSON, 2000).

3.1.3 Encapsulamento

Encapsulamento é uma técnica de projeto de software na qual os dados e procedimentos relacionados a esses dados são empacotados dentro de uma única entidade (WILSON,

2000).

Asseveram Horstmann e Cornell (2001) que a chave para a implementação desta especificidade é projetar programas que não tenham acesso direto às variáveis de instância de uma classe. Assim, os programas devem interagir com seus dados somente através dos métodos do objeto.

Pode-se considerar um carro como exemplo elucidativo de encapsulamento. Um carro possui certas propriedades, tais como a velocidade em que trafega, por exemplo. Um carro também possui métodos². Por exemplo, pode-se usar o pedal do acelerador para aumentar a propriedade da velocidade deste carro. Configura-se, assim, o método `acelerar`.

O método `acelerar` pode aceitar um parâmetro que especifica quão rápido ou lento o carro pode se mover. No caso de um carro real, *o quanto* é expresso pela pressão que o motorista exerce no pedal do acelerador. Seguindo a linha do exemplo, o pedal do acelerador é parte integrante do carro, ou seja, ele está encapsulado dentro do objeto `carro`.

Os benefícios do encapsulamento, nos domínios da programação OO, podem não ser tão claros. Supondo-se a criação de um programa de desenho, é previsível a necessidade de se manter um histórico da localização das formas na página, sua largura e do tipo da forma (círculo, retângulo, polígono, etc). Sem estruturas de dados encapsuladas, será preciso criar várias estruturas separadas - tais como tabelas, listas ou vetores - para armazenar toda a informação necessária. Uma lista pode armazenar referências para cada uma das formas que o usuário criar. Uma outra estrutura pode armazenar a localização, outra a cor, e assim por diante.

Desta maneira, cada vez que uma forma for adicionada pelo usuário, o programa deve adicionar suas informações em cada uma das estruturas. A exclusão de uma forma deve envolver a exclusão da correspondente entrada de cada lista. Supondo-se o uso destas formas em outro programa, seria necessária a seleção de todas as estruturas de dados relacionadas às formas e a sua inclusão no outro programa. A Figura 16 compara a abordagem encapsulada com a não encapsulada.

Com o encapsulamento, há a possibilidade de se projetar um objeto `forma` que encapsula todos os dados (propriedades) e procedimentos (métodos) relacionados a uma determinada forma. Pelo fato do objeto `forma` conter todas as informações necessárias para uma forma, pode-se considerar que o objeto foi projetado para ser facilmente reuti-

²O conceito de métodos será apresentado na seção 3.1.9

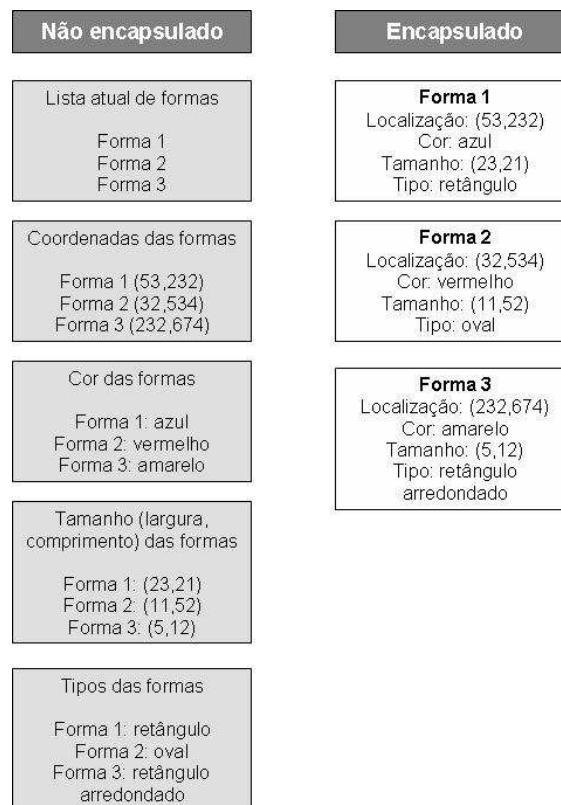


Figura 16: Duas abordagens para representar formas: não encapsulada e encapsulada
 Fonte: Wilson (2000)

lizado em outro programa.

3.1.4 Ocultação de informação

Ainda utilizando um carro como objeto comparativo entre o mundo real e o paradigma de orientação a objeto, um motorista não precisa preocupar-se sobre como a energia aplicada no pedal do acelerador aumenta a velocidade do carro. Em suma, pode-se dirigir um carro muito bem sem saber o que ocorre para que isso seja possível. Isto permite que se compare um carro com uma *caixa-preta*.

Assim, a facilidade de se usar um método sem tomar conhecimento dos detalhes de sua implementação é uma característica chamada ocultação de informação, cujo conceito foi introduzido com a programação estruturada e é um benefício natural do conceito de encapsulamento do paradigma OO (WILSON, 2000; DEITEL; DEITEL, 2002).

Supondo que a linha de pseudo-código a seguir aumente a velocidade do carro em 10Km/h,

```
carro.acelera(10)
```

estrutura semelhante aumentaria a velocidade em 25km/h, como segue:

```
carro.acelera(25).
```

Desta maneira, não há a necessidade de se aprofundar no conhecimento das linhas de código para entender ou descobrir como o método funciona. Isto fica bem claro, mesmo sem a explicitação dos detalhes da implementação do método `acelera` e facilita o reuso de objetos eventualmente criados por outro programador.

A reutilização de um método pode-se dar pelo seu aproveitamento em relação a outro objeto. Por exemplo, o método `acelera` pode ser utilizado para representar a variação de velocidade do objeto `motocicleta`, guardando as mesmas características de sua utilização em relação a `carro`.

Linguagens orientadas a objeto geralmente fornecem mecanismos para separar a interface do programador (a parte do objeto que é visível a alguém que esteja reutilizando um objeto) das partes internas do objeto, criadas pelo programador deste. Por conta da ocultação dos detalhes da implementação em relação a um usuário do objeto, é menos provável que um programador faça alterações inadvertidas em partes do código do objeto que está sendo reutilizado.

Para que a ocultação de informação ocorra, os objetos devem separar suas funções internas das partes que os conectam a outros objetos. As funções internas são chamadas *implementação* e a parte que conecta os objetos a outros é chamada *interface*. Um grande benefício da programação orientada a objeto é facilitar a criação de código reusável (WILSON, 2000). Modularidade significa que um programa é projetado para particionar uma grande tarefa de programação em sub-tarefas, as quais são codificadas independentemente e depois montadas para se obter o programa completo. Com um bom projeto modular, os módulos são projetados para minimizar a dependência uns em relação aos outros.

Por sua vez, na programação estruturada, a modularidade é obtida através da criação de subrotinas conhecidas como procedimentos ou funções. Para manter controle das interações entre as subrotinas, os dados são passados de ou para a subrotina através de parâmetros. Pelo encapsulamento, programas orientados a objeto fornecem um grau ainda maior de modularidade, já que os dados necessários e subrotinas estão incluídos no objeto.

3.1.5 Classes

Classe é uma construção que descreve comportamento e atributos comuns. Em outras palavras, é uma estrutura de dados que inclui dados e funções. Na mesma linha de raciocínio, classe pode ser encarada como um código de programação que define um modelo ou padrão para criação de um tipo particular de objeto (WILSON, 2000).

Linguagens orientadas a objetos permitem a criação e manipulação destas estruturas de dados. Cada modelo (classe) define um tipo específico de objeto, o qual tem certos atributos. Uma vez definida a classe, pode-se então criar um número qualquer de instâncias daquele objeto.

Uma instância é um objeto particular que foi criado a partir de uma classe. Por sua vez, instanciação é o processo ou ato de criação de um objeto baseado em uma classe (WILSON, 2000).

Considerando uma linguagem de orientação a objeto fictícia, pode-se criar inúmeras instâncias de um objeto. A Figura 17 apresenta o código de criação de duas instâncias de uma classe, na linguagem fictícia.

```
1 // As linhas seguintes definem uma classe
2 classe pessoa {
3   string nome;
4   inteiro idade;
5   string corCabelo;
6   string corOlho;
7   string localizacaoAtual;
8 }
9
10 // As linhas seguintes instanciam objetos de uma classe
11 pessoa1 = pessoa.cria();
12 pessoa2 = pessoa.cria();
13
14 // As linhas seguintes atribuem propriedades aos objetos
15 pessoa1.nome = "Luiz Estácio";
16 pessoa1.localizacaoAtual = "Brasília";
17 pessoa2.nome = "Fernando Elite"
```

Figura 17: Exemplo de criação de duas instâncias de uma classe

Da linha 2 até a 8 define-se a classe `pessoa`, com os atributos `nome`, `idade`, `corCabelo`, `corOlho` e `localizacaoAtual`. Assim, uma pessoa possui um nome (que supõe-se seja composto por uma cadeia de caracteres), uma idade (que pode conter um número inteiro) e assim por diante.

As linhas 11 e 12 instanciam, cada uma, um objeto `pessoa`, atribuindo cada novo objeto a um nome único (`pessoa1` e `pessoa2`). As linhas 15, 16 e 17 atribuem propriedades aos objetos que foram instanciados.

3.1.6 Subclasses

Considerando a classe codificada na Figura 18, define-se `mano1` e `mano2` como tipos especiais de `pessoa`, através da instanciação desta classe (WILSON, 2000).

```
1 classe pessoa {
2 string nome;
3 inteiro idade;
4 string corCabelo;
5 string corOlho;
6 }
7
8 mano1 = pessoa.cria();
9 mano2 = pessoa.cria();
10
11 mano1.nome = "Luiz Estácio";
12 mano2.nome = "Fernando Elite";
```

Figura 18: Exemplo de criação de subclasse

Além da classe `pessoa`, é possível criar uma classe inteiramente nova, conforme mostrado na Figura 19:

```
1 classe pessoa {
2 string nome;
3 inteiro idade;
4 string corCabelo;
5 string corOlho;
6 }
7
8 classe mano {
9 string nome;
10 inteiro idade;
11 string corCabelo;
12 string corOlho;
13 string cidadeAtual;
14 string pratoPredileto;
15 inteiro quantosAnosEhAmigo;
16 string filmePredileto;
17 string livroQueEstaLendo;
18 }
```

Figura 19: Exemplo de criação de nova classe

Pode-se afirmar que esta construção é ineficiente, levando-se em conta que a classe `pessoa` e a classe `mano` têm diversos atributos em comum. Supondo-se a necessidade de adicionar mais atributos a `pessoa` (além de nome, idade, `corCabelo` e `corOlho`), tais atributos adicionais não seriam incluídos automaticamente na classe `mano`.

No entanto, a maioria das linguagens de programação orientadas a objetos fornecem meios de criação de uma classe de objetos baseada em outra classe ou em um subconjunto desta. Tal característica é chamada de subclasse. O código mostrado na Figura 20 mostra como uma subclasse pode ser criada a partir de uma classe, numa linguagem fictícia.

```
1 classe pessoa {
2   string nome;
3   inteiro idade;
4   string corCabelo;
5   string corOlho;
6 }
7
8 classe mano baseada em pessoa {
9   string cidadeAtual;
10  string pratoPredileto;
11  inteiro quantosAnosEhAmigo;
12  string filmePredileto;
13  string livroQueEstaLendo;
14 }
```

Figura 20: Exemplo de criação de uma subclasse

Neste exemplo, a classe `mano` é uma subclasse da classe `pessoa`. Isto significa que a subclasse possui todos os atributos da classe `pessoa`, bem como todos os atributos que foram adicionados.

3.1.7 Reutilização de classes

Uma das características mais úteis das linguagens OO é a capacidade de facilitar a reutilização de código, ou seja, o aproveitamento de classes e seus métodos já escritos. Tal particularidade diminui a necessidade de escrever novos métodos e classes, diminuindo o retrabalho e a possibilidade da inserção de novos erros no código (SANTOS, 2003).

Enquanto as linguagens procedurais implementam reutilização de código através de funções que podem ser chamadas a partir de vários programas diferentes, as linguagens orientadas a objeto permitem a criação de classes baseadas em outras classes, que podem conter métodos das classes originais e métodos próprios.

Existem dois mecanismos básicos de reutilização de classes: delegação (ou composição) e herança.

3.1.7.1 Agregação ou composição

Uma forma de se implementar reuso de software é a composição, onde uma classe possui referências a objetos de outras classes como membros (DEITEL; DEITEL, 2002).

Este mecanismo permite a criação de novas classes ao se incluir uma instância de uma classe como um de seus atributos (SANTOS, 2003). Como exemplo, pode-se tomar a classe `pessoa`, descrita na Figura 18 e a classe `mano`, mostrada na Figura 19. A criação de um atributo do tipo `pessoa` na classe `mano` configuraria a composição do tipo `pessoa`

em relação ao tipo `mano`.

3.1.7.2 Herança

Uma dos benefícios de se poder criar uma subclasse é que esta herda os atributos de sua classe pai. Este é um dos mais poderosos aspectos da programação orientada a objetos. Através da herança, pode-se fazer uma alteração na classe pai que será aplicada a todas as subclasses baseadas na classe pai. Além disto, os atributos podem ser passados para múltiplas gerações.

No exemplo mostrado na Figura 21 pode-se notar duas gerações, sendo que `cachorro` herda os atributos de `animal` e `bigle` herda os atributos de `cachorro` e `animal`.

```
1 classe animal {
2 inteiro idade;
3 string reino;
4 string filo;
5 string classe;
6 string ordem;
7 string familia;
8 string genero;
9 string especie;
10 }
11
12 classe cachorro baseada em animal {
13 string raca;
14 string marcas;
15 }
16 classe bigle baseada em cachorro {
17 string idOficial;
18 }
```

Figura 21: Exemplo de herança no paradigma OO.

3.1.8 Polimorfismo

O paradigma de orientação a objeto permite que se trate diferentes objetos de maneiras similares. Por exemplo, uma classe `carro` poderia ser passada para uma função chamada `move`, a qual permite que um carro se mova em certa direção e velocidade. Neste mesmo contexto, supõe-se a existência de uma classe `helicoptero` que deverá ser passada também à função `move`.

Sabe-se, no entanto, que veículos não se movem necessariamente da mesma maneira. Com a programação orientada a objetos, há a possibilidade de se criar abstrações – tais como a função `move` – que ocultam tais diferenças e permitem que as classes tenham um comportamento similar, de tal forma que carros e helicópteros possam se mover sem que haja preocupação com detalhes escondidos pela abstração.

Considerando o código mostrado na Figura 22, o comando `move(carro1, 75, "norte")` faz o carro mover-se a 75Km/h na direção norte, assim como `move(helicoptero1, 75, "norte")` faz o helicóptero mover-se na mesma direção e velocidade.

```
1 classe veiculo {
2 // propriedades e métodos de veiculo são escritos aqui
3 }
4
5 classe carro baseada em veiculo {
6 // propriedades e métodos de carro são escritos aqui
7 }
8
9 classe helicoptero baseada em veiculo {
10 // propriedades e métodos de helicoptero são escritos aqui
11 }
12
13 funcao move (veiculo oVeiculo, inteiro aDistancia, direcao aDirecao) {
14 se oVeiculo é um helicoptero
15   mova-o como um helicoptero
16 se oVeiculo é um automovel
17   mova-o como um automovel
18 }
19
20 helicoptero1 = novo helicoptero;
21 carro1 = novo carro;
22 move(helicoptero1, 75, "norte");
23 move(carro1, 75, "norte");
```

Figura 22: Exemplificação de polimorfismo

Este código define uma classe `veiculo` e em seguida cria duas subclasses de `veiculo`: `carro` e `helicoptero`, que herdam as propriedades e métodos da classe pai. A função `move` é também declarada (linha 13) e seu funcionamento baseia-se na passagem de três parâmetros, que são um objeto de veículo, um inteiro que especifica a velocidade e a direção. A função `move` executa rotinas diferentes para ajustar a velocidade e direção de diferentes tipos de veículos.

Finalmente, o código apresentado cria dois objetos: `helicoptero1`, na linha 20 e `carro1`, na linha 21, os quais compartilham propriedades da classe pai `veiculo`. O código, então, usa a mesma função para mover ambos os tipos de objetos, apesar da função tratar os dois objetos de forma diferente. Mover um helicóptero ficou sendo uma ação similar a mover um carro, caracterizando, assim, o polimorfismo.

3.1.9 Métodos

A programação orientada a objeto encapsula dados (atributos) e métodos (comportamentos) dentro dos objetos. Os dados e métodos de um objeto estão intimamente ligados (DEITEL; DEITEL, 2002).

Em linguagens procedimentais, a programação tende a ser orientada a ação e ter a

função como unidade de programação. No paradigma de orientação a objeto, as funções não desaparecem, mas são encapsuladas como métodos com os dados que eles processam dentro das classes. Tais dados são chamados atributos (DEITEL; DEITEL, 2002).

Santos (2003) compara classes a modelos contendo dados e métodos às operações que manipulam tais dados. As operações relacionadas aos modelos - denominadas métodos, seguem regras para sua criação nas diversas linguagens de programação em que são implementados. A esse respeito, de modo geral, Santos (2003) salienta que:

- métodos não podem ser criados dentro de outros métodos, nem fora de uma classe, de forma isolada;
- nomes de métodos devem refletir a essência de sua operação. Embora isto não se constitua propriamente em uma regra, tornou-se uma boa prática de programação ao longo dos anos.

Os métodos possuem características de segurança que os tornam acessíveis a partir de outras classes ou somente a partir de métodos da mesma classe, sendo assim conhecidos como métodos públicos e privados, respectivamente. Segundo Horstmann e Cornell (2001), a razão para tal distinção deve-se ao fato de que o programador, para implementar certas operações, pode desejar particionar o código em vários métodos separados, muitos dos quais podem não ter significância ou utilidade para outros programadores. Tais métodos são melhor implementados como métodos privados.

Outra característica atribuída aos métodos pela maioria das linguagens de programação OO é a de torná-los estáticos. Os métodos estáticos pertencem a uma classe e não operam em nenhuma instância de uma classe. Isso significa que se pode usá-los sem criar uma instância de uma classe (HORSTMANN; CORNELL, 2001).

De forma a exemplificar tais conceitos, a Figura 23 mostra o código de uma classe Java que contém métodos e atributos relativos à manipulação de datas.

As linhas 2 e 3 executam a declaração dos atributos da classe `DataSimples`. Embora não estejam presentes no exemplo, atributos específicos para cada método da classe poderiam ser declarados localmente.

O método `inicializaDataSimples` recebe argumentos para inicializar os atributos da classe `DataSimples`. Este método chama o método `dataÉVálida` para verificar se os argumentos são correspondentes a uma data válida. Caso sejam, inicializa os atributos. Caso contrário, inicializa os três atributos com valor igual a zero.

O método `dataÉválida` recebe três valores como argumentos e verifica de maneira simples se os dados correspondem a uma data válida. Caso seja, retorna a constante booleana `true`. Caso contrário, retorna a constante booleana `false`.

Sugere Santos (2003) que este algoritmo seja bastante simples e não tem verificada sua exatidão, servindo apenas como ilustração para os conceitos de métodos e atributos.

```
1 classe DataSimples {
2 inteiro dia, mes;
3 inteiro ano;
4 nulo inicializaDataSimples(byte d, byte m, short a)
5 {
6     se (dataÉválida(d,m,a)) entao
7     {
8         dia = d; mes = m; ano = a;
9     }
10    senao
11    {
12        dia = 0; mes = 0; ano = 0;
13    }
14 }
15 booleano dataÉválida(byte d, byte m, short a)
16 {
17     se ((d >=1) &&
18         (d <=31) &&
19         (m >=1) &&
20         (m <=12))
21         retorna verdadeiro;
22     else
23         retorna falso;
24 }
25 booleano éIgual(Data Simples outraDataSimples)
26 {
27     se ((dia == outraDataSimples.dia) &&
28         (mes == outraDataSimples.mes) &&
29         (ano == outraDataSimples.ano))
30         retorna verdadeiro;
31     else
32         retorna falso;
33 }
34 nulo mostraDataSimples()
35 {
36     imprime(dia);
37     imprime("/");
38     imprime(mes);
39     imprime("/");
40 }
41 }
```

Figura 23: Exemplo de classe Java contendo métodos e atributos
Fonte: Santos (2003)

3.1.10 Campos

As variáveis de um tipo classe são seus campos. Variáveis de classe (*static*) são em número de uma por classe. Variáveis de instância são em número de uma por instância da classe. Campos podem ser modificados usando-se modificadores de acesso e seus efeitos estendem-se às superclasses e superinterfaces da classe. Uma classe herda de sua super-

classe direta todos os seus campos que não são ocultos por uma declaração expressa na classe.

3.1.10.1 Modificadores de campo

Campos podem ser declarados públicos, privados ou protegidos. Um campo público pode ser acessado por qualquer código Java. Um campo privado por ser acessado apenas no corpo da classe que contém sua declaração. Um campo que não é declarado público, protegido ou privado tem acesso *default* e pode ser acessado unicamente a partir do pacote em que foi declarado.

Um campo protegido pode ser acessado apenas pelo código responsável pela implementação daquele campo. Em outras palavras, um campo protegido pode ser acessado a partir de qualquer ponto do pacote em que foi declarado e, além disso, pode ser acessado a partir de qualquer declaração que se encontre numa subclasse do classe que contém sua declaração (LINDHOLM; YELLIN, 1996).

Se um campo é declarado `static`, passa a existir exatamente uma materialização do campo, não importando quantas instâncias da classe possam eventualmente ser criadas. Segundo Lindholm e Yellin (1996), um campo estático, por vezes chamado de variável de classe, torna-se existente quando a classe é inicializada.

Este capítulo tratou de aspectos do paradigma OO, principalmente aqueles com relação direta com a ferramenta construída. Na seqüência, será oferecida a descrição completa da ferramenta desenvolvida para este trabalho, incluindo a tecnologia utilizada para este fim.

4 Ferramenta de Visualização Tridimensional de Programas Orientados a Objeto

Este capítulo trata da ferramenta construída para visualização de programas OO e das atividades que auxiliaram nesta construção. Seu código-fonte atual é investigado em suas partes mais relevantes, classe a classe, incluindo apresentação do diagrama de classes. Antes do detalhamento da ferramenta são apresentadas as tecnologias utilizadas para a construção da ferramenta e as metáforas representativas dos componentes de um programa.

Com o intuito de formar base comparativa entre a visualização de programas em ambiente bidimensional e tridimensional, a Figura 24 apresenta o Diagrama de Classes do programa que implementa a ferramenta.

As diferenças na análise de uma representação bidimensional de um programa em relação à visualização tridimensional serão abordadas adiante, no Capítulo 5.

4.1 Tecnologias utilizadas

Esta seção destina-se à apresentação da base tecnológica que viabilizou a construção da ferramenta. Utilizando-se da possibilidade de reuso de código proporcionado pela linguagem de programação Java, as classes próprias da ferramenta fazem referência a outras classes que viabilizam o reconhecimento dos componentes do programa sob análise e a apresentação destes em formato tridimensional.

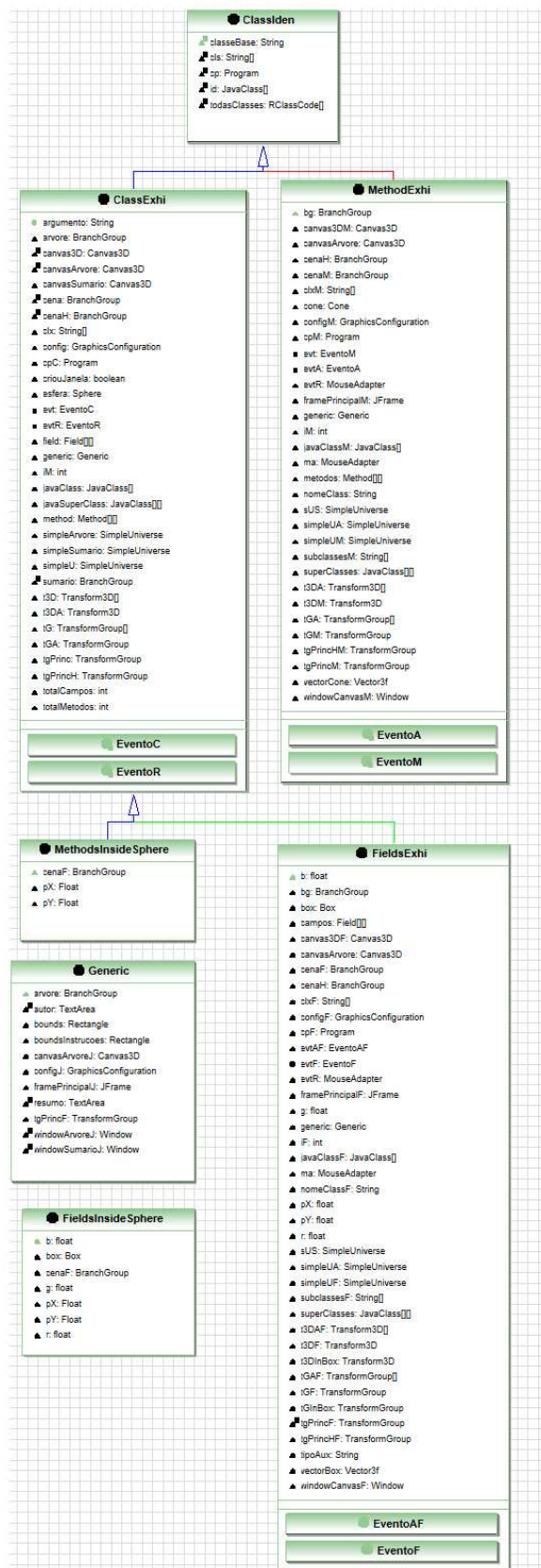


Figura 24: Diagrama de classes da ferramenta

4.1.1 API Java 3D

A API Java 3D é uma interface destinada a escrita de programas próprios para a exibição de objetos tridimensionais. Ela fornece uma coleção de classes que viabilizam a construção e manipulação de geometria tridimensional, além de fornecer meios de renderização desta geometria. A API é projetada com a flexibilidade necessária para que se possa criar universos virtuais precisos com uma grande variedade de tamanhos. Os detalhes da renderização são tratados automaticamente pela API, que tira vantagem das *threads* Java para executá-la em paralelo (BOUVIER, 2000).

Um programa Java 3D cria instâncias de objetos Java 3D e os coloca em grafos de cena. O grafo de cena é um arranjo de objetos tridimensionais em uma estrutura de árvore, que especifica por completo o conteúdo de um universo virtual, incluindo sua renderização (BOUVIER, 2000). Através de comandos de alto nível, é possível exibir objetos primitivos, tais como esferas, cones e cubos, além de atribuir cores, textura e iluminação a estas formas.

Todo programa Java 3D é, ao menos parcialmente, montado a partir de objetos do conjunto de classes da API. Esta coleção de objetos descreve um universo virtual. Embora a API conte com mais de 100 classes, um pequeno número delas é necessário para que se construa um universo virtual simples que conte, inclusive, com animação (BOUVIER, 2000).

Um universo virtual é criado a partir de um grafo de cena (BOUVIER, 2000). Um grafo de cena é criado usando-se instâncias das classes Java 3D e montado a partir de objetos que definem a geometria, luz, posição, orientação e aparência das formas em tela. É comum definir-se grafo de cena como uma estrutura de dados composta por nós e arcos. Genericamente, um nó é um dado elementar e um arco representa um relacionamento entre os dados. Num grafo de cena, os nós representam instâncias de classes Java 3D e os arcos representam relacionamentos entre as instâncias. O relacionamento mais comum é o pai-filho. Um nó de grupo é aquele que pode ter qualquer número de filhos, mas apenas um pai. Um nó folha pode ter apenas um pai e nenhum filho. O outro relacionamento é uma referência. Uma referência associa um objeto `NodeComponent` com um nó do grafo de cena e define os atributos de geometria e aparência usados para renderizar os objetos visuais.

Os grafos de cena são construídos por objetos `Node` em relacionamentos do tipo pai-filho, formando uma estrutura de árvore. Nela, um nó é a raiz e os outros nós são acessíveis

percorrendo-se arcos a partir da raiz. A Figura 25 mostra um exemplo de grafo de cena.

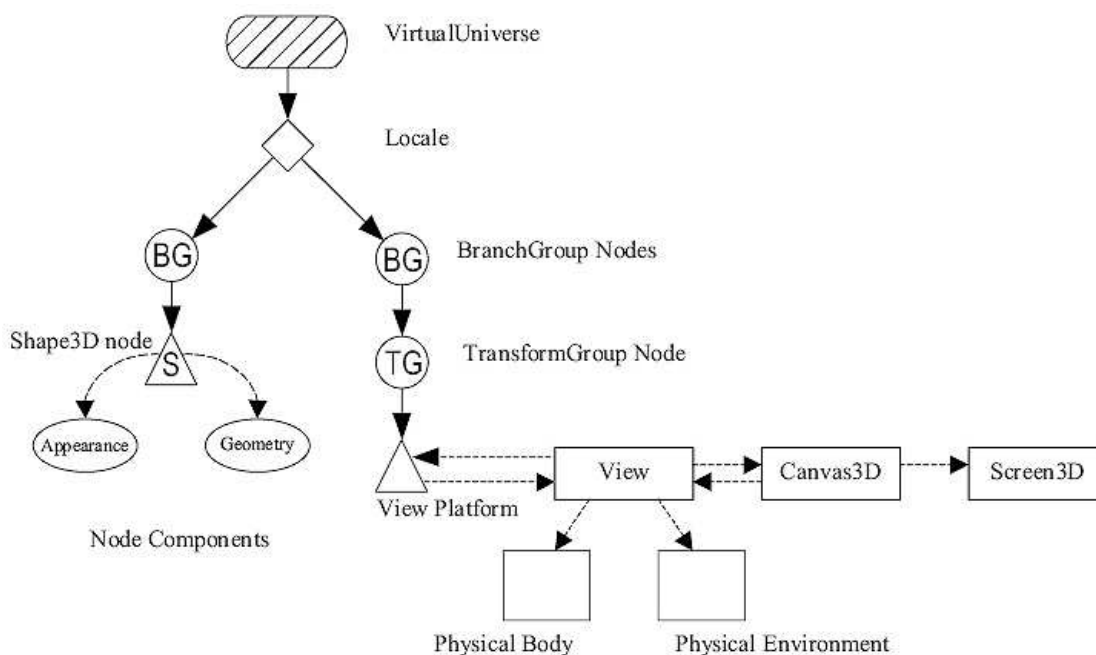


Figura 25: Exemplo de grafo de cena
Fonte: Bouvier (2000)

Através deste exemplo, pode-se identificar apenas um caminho entre a raiz e cada um dos nós folha. Assim, os arcos de uma árvore não formam ciclos e cada caminho no grafo de cena especifica por completo as informações da sua folha, o que inclui a localização, orientação e tamanho do objeto visual. Um grafo de cena é formado a partir de árvores cujas raízes são objetos *Locale*. Os objetos *NodeComponents* e os arcos de referência não fazem parte da árvore (BOUVIER, 2000).

Representações gráficas de um grafo de cena servem normalmente como ferramenta de projeto e documentação de programas Java 3D. A Figura 26 traz o grafo de cena contendo instâncias de classes Java 3D presentes em uma das classes da ferramenta, com o intuito apenas de transmitir a idéia de como um grafo desta natureza é composto.

O objeto *cena* é instância da classe *BranchGroup*. Um *BranchGroup* atua como um ponteiro para a raiz do grafo de cena e nele são anexados, por exemplo, as formas presentes em tela e suas propriedades visuais, tais como luz e cor. Um *BranchGroup* deve conter também elementos que controlam o comportamento dos objetos. No caso específico desta parte da ferramenta, trata-se do zoom, rotação e translação dos objetos.

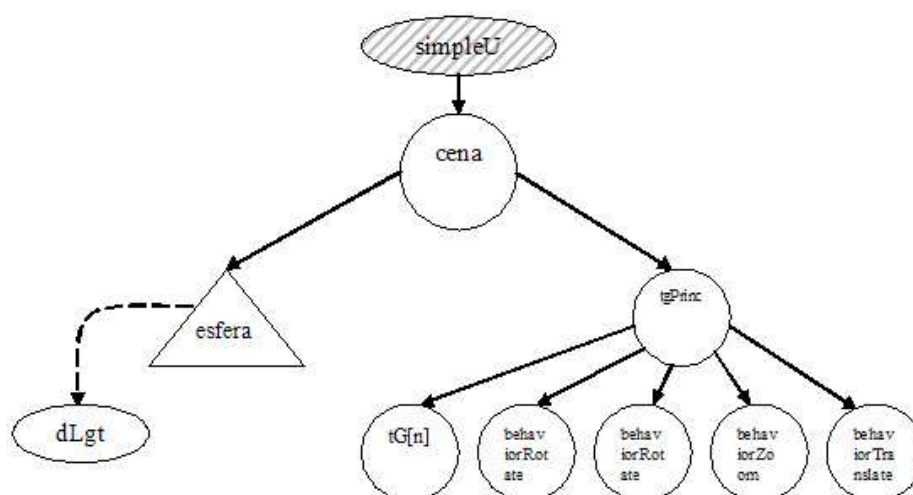


Figura 26: Grafo de cena de uma das classes da ferramenta

4.1.2 Pacote lookup

O pacote `lookup` é parte integrante da ferramenta JaBUTi, desenvolvida por Vincenzi et al. (2003)¹. As classes do `lookup` – `Program`, `RClassCode`, `RClass` e `ClassClosure` – são usadas neste projeto para analisar a estrutura do programa submetido à ferramenta.

A principal classe deste pacote é a `Program`, que representa a estrutura hierárquica do programa (DELAMARO, 2005). A criação de um objeto do tipo `Program` requer quatro parâmetros: o primeiro refere-se ao nome da classe base, a partir da qual deve ser construída a estrutura do programa. O segundo, quando *true*, mantém as classes da API Java fora da análise da ferramenta. O terceiro parâmetro refere-se à lista de pacotes ou classes que não devem fazer parte da estrutura. O quarto refere-se ao caminho onde se encontram os arquivos `.class` das classes que compõem o programa (DELAMARO, 2005).

A classe `ClassClosure` armazena o conjunto de classes que fazem parte do programa. Para cada classe identificada como parte da estrutura, é criado um objeto do tipo `RClass` ou `RClassCode`, dependendo se o arquivo `.class` encontra-se no caminho fornecido. Segundo Delamaro (2005),

um objeto `RClass` possui apenas informações de quem são as suas subclasses ou as classes que a implementam (no caso de uma interface). Já um objeto `RClassCode` possui informação sobre quem é sua superclasse, quem são as interfaces por ela implementadas e todas as informações sobre a classe em si, obtida a partir do arquivo `.class`.

A análise de um arquivo de classe Java se dá através de uma biblioteca desenvolvida

¹A JaBUTi será descrita de forma mais detalhada no Capítulo 6

pela Apache Software Foundation e descrita a seguir.

4.1.3 Biblioteca BCEL

A *Byte Code Engineering Library* (BCEL) é uma biblioteca projetada para fornecer ao usuário a possibilidade de analisar, criar e manipular arquivos de classes Java, representadas por objetos que contêm toda a informação simbólica da classe: métodos, campos e instruções em `bytecode`.

Segundo Apache (2002-2003), a BCEL tem sido usada com sucesso em diversos projetos tais como compiladores, otimizadores, geradores de código e ferramentas de análise.

A principal classe utilizada na ferramenta é a `JavaClass`, através da qual identifica-se, além da própria classe, os métodos e atributos desta. Tais componentes do programa são identificados a partir do arquivo `.class` da classe e, uma vez lidos deste arquivo, podem ser transformados pela aplicação e reescritos em um novo arquivo `.class` (GREEN, 2006; APACHE, 2002-2003).

Na Figura 27 é exibido o esquema representativo da hierarquia entre bibliotecas de classes presentes na construção da ferramenta objeto deste trabalho.

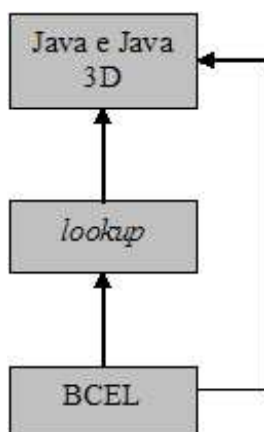


Figura 27: Hierarquia entre bibliotecas implementadas na ferramenta do projeto

O pacote `lookup` cria instâncias de classes da biblioteca BCEL para construir a estrutura do programa sob análise. No entanto, algumas classes da BCEL são instanciadas diretamente pelo código da ferramenta, conforme detalhado na seção 4.4.

4.2 Metáforas utilizadas

A escolha das metáforas utilizadas neste projeto deu-se com base em sua simplicidade e facilidade de implementação. Esferas, cubos e cones são formas geométricas implementadas diretamente pela API Java 3D, não havendo necessidade de construção destas formas através de união de pontos em coordenadas previamente fornecidas. Neste sentido, diz-se que esferas, cubos e cones são formas primitivas da API.

A Figura 28 mostra as metáforas utilizadas na ferramenta.



Figura 28: Metáforas utilizadas na ferramenta

A transmissão ao usuário da idéia contextual do programa através das metáforas escolhidas é imediata. O benefício da utilização de recursos como cores, transparência, tamanhos e formas diferentes para cada tipo de componente abrange a compreensão do contexto geral do programa e de certas situações particulares.

Como exemplo de visualização geral, pode-se citar a cena gerada pela execução inicial do programa². Esta cena, retratada na Figura 29, é composta por pares de esferas representativas das classes em cores diferentes entre si, no intuito de destacá-las umas das outras. A colocação de pares de esferas (ao invés de se representar cada classe com uma única esfera) serve para que uma esfera represente os métodos da classe e a outra represente os atributos da classe.

Ainda neste contexto, merece destaque a colocação de objetos representativos de métodos e atributos no interior das esferas, possibilitada através da aplicação da transparência nas esferas. De imediato, o usuário pode observar a quantidade de métodos e atributos em cada classe, sem mesmo avançar na execução da ferramenta. Para tanto, o usuário conta ainda com a distinção de tamanho entre cada esfera: quanto maior o número de métodos/atributos, maior a esfera que os contém.

Certas situações em que a visualização foca contextos particulares também tiram

²Os aspectos de operação e de implementação serão discutidos nas seções 4.3 e 4.4, respectivamente.

vantagem da utilização de cores nas metáforas. Nas seções seguintes, serão abordados tais contextos, juntamente com aspectos operacionais relacionados às metáforas.

4.3 Aspectos operacionais

A ferramenta desenvolvida está apta a reconhecer, dada uma certa classe de entrada, todas as classes que guardem correspondência com aquela entrada. Uma vez estabelecidas as classes que são efetivamente relacionadas com a classe de entrada (ou classe base) e excluídas aquelas que não guardam vínculo direto com ela, é possível obter todos os métodos que fazem parte de todas as classes reconhecidas. Analogamente, os atributos de cada classe também podem ser manipulados e exibidos. Classe, método e atributo, além de certos relacionamentos entre classes, são os componentes de um programa orientado a objeto abordados pela ferramenta.

Entende-se como classe relacionada com a de entrada aquela classe construída pelo programador e que faz parte do sistema, excluindo-se classes previamente construídas e que são instanciadas pelo sistema sob análise.

Ao ser executado o programa, são exibidas na parte 5 da imagem as classes reconhecidas pela ferramenta que relacionam-se com a classe de entrada. Uma classe é representada por um par de esferas, sendo a primeira uma representação dos métodos da classe e a segunda uma representação dos atributos desta classe. Cada par de esferas recebe o nome da classe que retratam e têm tamanho variável conforme a quantidade de métodos e atributos desta classe. Assim, uma esfera que representa uma classe com muitos métodos terá tamanho maior que uma esfera que represente uma classe com poucos métodos. O mesmo vale para a esfera que representa os atributos.

Para efeito de exemplificação de execução, a classe `ClassIden.java` foi eleita como entrada. A Figura 29 mostra a execução inicial da ferramenta. As diversas partes da imagem são numeradas para facilitar a explicitação de suas funções.

Objetivando melhorar sua distinção visual, cada par de esferas é colorido de forma distinta entre si, ou seja, um par de esferas sempre terá cor diferente de outro. Além disso, a cada execução da ferramenta, as cores das esferas são geradas aleatoriamente pela classe que as constrói. A visualização das classes é apresentada na parte 5 da imagem, em sua porção maior.

Na parte 1, a imagem apresenta as metáforas usadas pela ferramenta e logo abaixo,

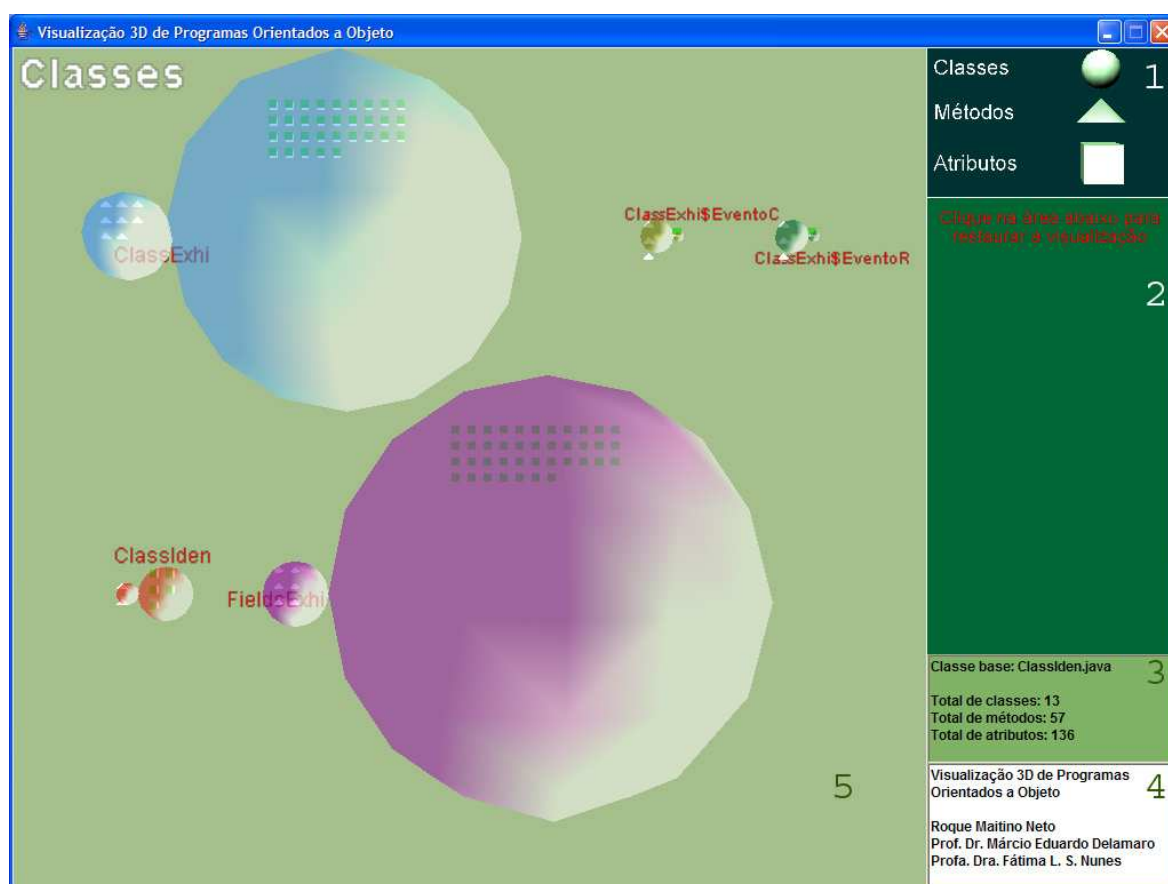


Figura 29: Execução inicial da ferramenta

na parte 2, segue área de retorno a visualização inicial das classes, abordada com detalhes na seção 4.3.1. Na parte 3, pode-se visualizar resumo textual do sistema. É apresentada a quantidade de classes relacionadas e as quantidades totais de métodos e atributos do sistema. Mais abaixo, a parte 4 apresenta o nome da ferramenta, o autor e os orientadores.

4.3.1 Interação com usuário

A interação da ferramenta com o usuário é realizada através da utilização do mouse. O acesso aos ambientes em que são detalhados os métodos e atributos das classes é feito através do acionamento do botão esquerdo do mouse sobre a classe a ser investigada. Tal ação leva o usuário da execução inicial a outro ambiente tridimensional, onde os métodos (ou atributos, dependendo da esfera acionada) são exibidos e nomeados também na parte 5 da imagem mostrada na Figura 30.

Assim, se o usuário optar em visualizar os métodos em detalhes, obterá um ambiente semelhante ao exibido na Figura 30, que retrata os métodos da classe `ClassExhi.java`.

O acionamento do mouse sobre a esfera que representa os atributos da classe leva ao



Figura 30: Detalhamento dos métodos da classe `ClassExhi.java`

detalhamento destes atributos, representados cada um por um cubo. A Figura 31 mostra em detalhe os atributos da classe `ClassExhi.java`.

Tal visualização permite ao usuário reconhecer cada método e cada atributo representados respectivamente por um cone e um cubo nomeado de forma individual. Acionando-se o botão direito do mouse sobre um cone pode-se observar a assinatura do método. A Figura 32 mostra a assinatura do método `exibeClasses`, da classe `ClassExhi.java`.

Neste caso, a assinatura é definida para identificar métodos no nível da Máquina Virtual Java (JVM). A sequência `(Ljava/lang/String; [Lorg/apache/bcel/classfile/Method; IFFFFF)V` vista na Figura 32 representa os tipos dos argumentos na ordem em que são dispostos: `String`, `Method`, `Integer`, `Float`, `Float`, `Float`, `Float`, `Float`. Por último, a letra `V` representa o tipo de retorno que, no exemplo, é `void`.

O acionamento do botão direito do mouse sobre um cubo mostra o tipo do atributo. Na Figura 33, um dos atributos da classe `ClassExhi.java` é tomado como exemplo.

A parte 2 da Figura 31 mostra o relacionamento hierárquico da classe selecionada em relação a outras do sistema, o que será tratado adiante nesta seção.

No caso específico da visualização detalhada dos atributos, estes são diferenciados por



Figura 31: Detalhamento dos atributos da classe ClassExhi.java

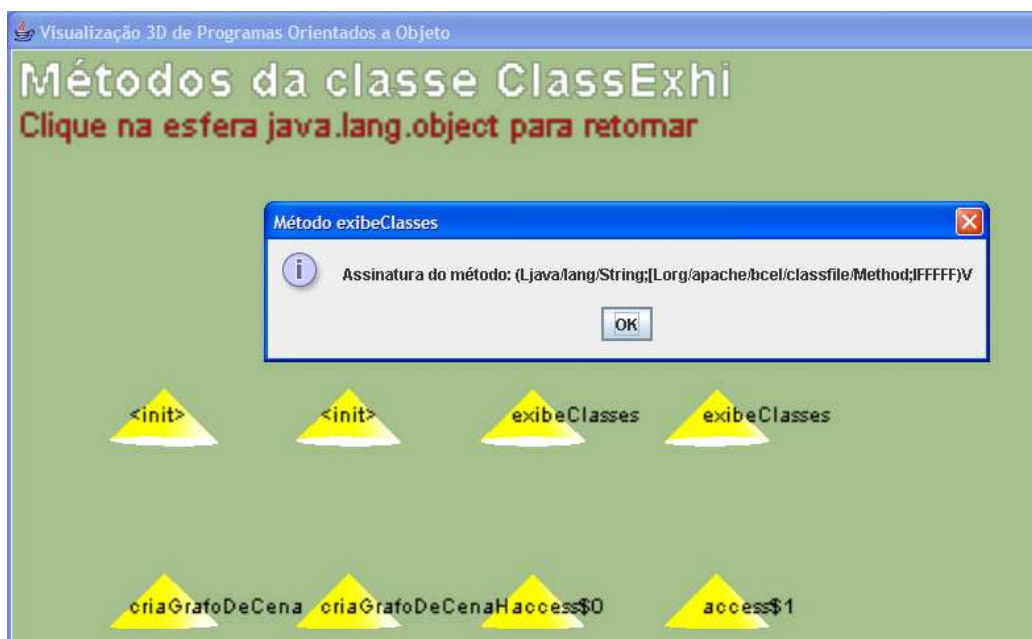


Figura 32: Assinatura do método exibeClasses

cores fixas, conforme seus modificadores de acesso e conforme a existência de agregação com alguma classe relacionada que compõe o sistema. Assim, atributos com modificador *Private* recebem a cor vermelha. Atributos *Public* são verdes e *Static* são azuis. Os que não possuem modificador são chamados *Package* e são coloridos de cinza claro. Já os cubos



Figura 33: Tipo do atributo argumento

amarelos representam atributos que compõem agregação com alguma classe relacionada. No interior do cubo é possível identificar um esfera que apenas representa tal classe, conforme mostra em detalhe a Figura 34.

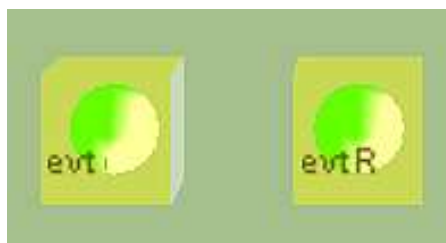


Figura 34: Detalhe de objetos que representam agregação

A legenda que relaciona cores com modificadores de acesso/agregação encontra-se na parte 1 da imagem.

Ao acionar o botão esquerdo do mouse sobre o cubo da agregação, o usuário visualiza os atributos da classe agregada. A imagem exibida na Figura 35 originou-se a partir do

acionamento do mouse sobre o atributo `generic`, pertencente à classe `ClassExhi.java`.



Figura 35: Detalhamento dos atributos da classe `Generic`, a partir do cubo da agregação

Conforme citado anteriormente, a porção direita da tela (parte 2) da ferramenta contém a exibição do relacionamento hierárquico da classe escolhida anteriormente para análise. Neste contexto, é possível observar as classes-pai e classes-filhas daquela escolhida. Além disso, o usuário pode escolher qualquer classe da hierarquia através do acionamento do mouse (exceto a própria anteriormente escolhida e a `java.lang.object`) para obter dela informações detalhadas de seus atributos ou métodos.

A ferramenta oferece ao usuário a possibilidade de interagir com os objetos (esferas, cones e cubos). Como termo de comparação, a Figura 36 mostra a execução da ferramenta em estado inicial, sem a aplicação de recursos de interação com o mouse.

As Figuras 37 e 38 mostram a aplicação de zoom e rotação, respectivamente, na exibição das classes do programa sob análise.

Através do botão esquerdo do mouse, pode-se rotacionar os objetos situados na parte 5 (Figura 35) da tela. Com o botão central, pode-se aplicar zoom à imagem e com o botão direito, move-se os objetos em qualquer direção. Os objetos e os textos que os nomeiam

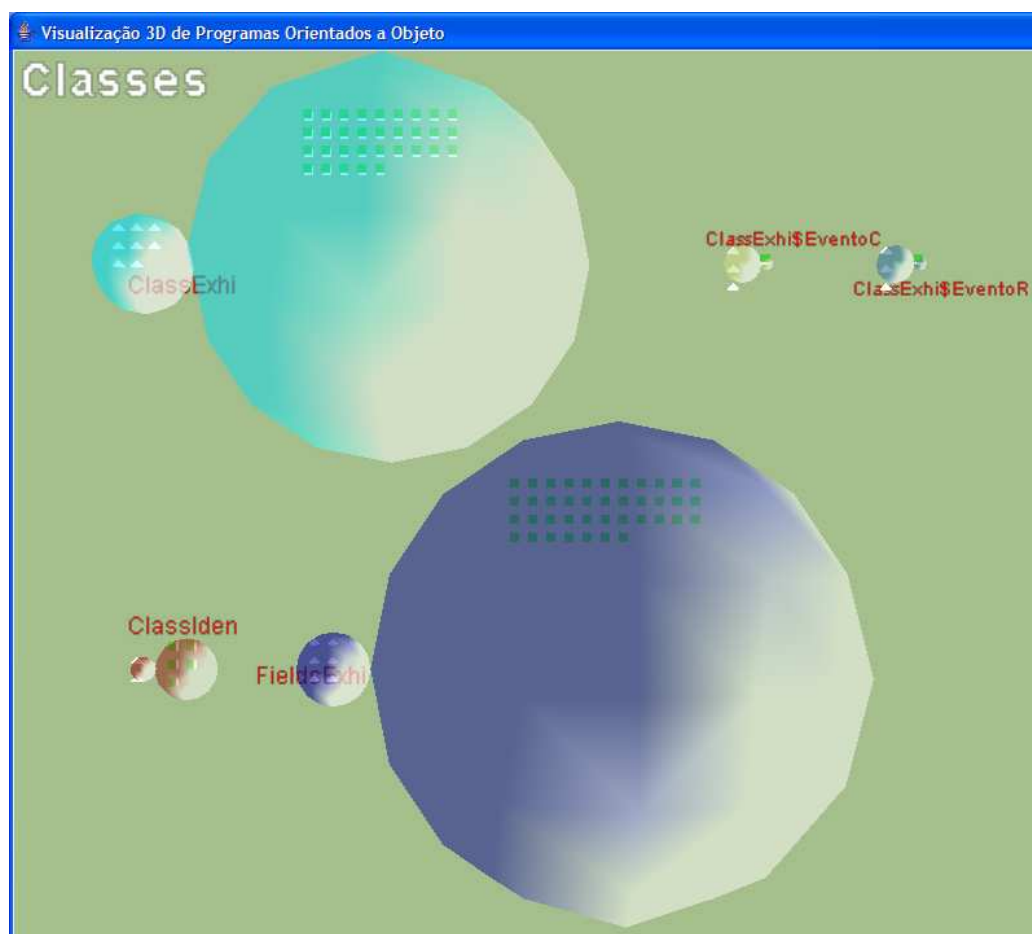


Figura 36: Execução inicial da ferramenta

são movimentados conjuntamente e não de forma individual.

Vale ressaltar que, dependendo do número de objetos a serem exibidos num mesmo contexto, a visualização pode não contemplá-los todos simultaneamente. Neste caso, é possível usar os recursos de rotação, zoom e translação para obter a visualização desejada.

4.4 Aspectos de implementação

Esta seção destina-se à descrição das partes mais relevantes do código-fonte da ferramenta. Cada subseção trata especificamente das classes que compõem o sistema, através da apresentação de seu código-fonte (em parte ou por completo) numerado e da descrição textual da sua execução. Cada fragmento de código será iniciado pela linha 1.

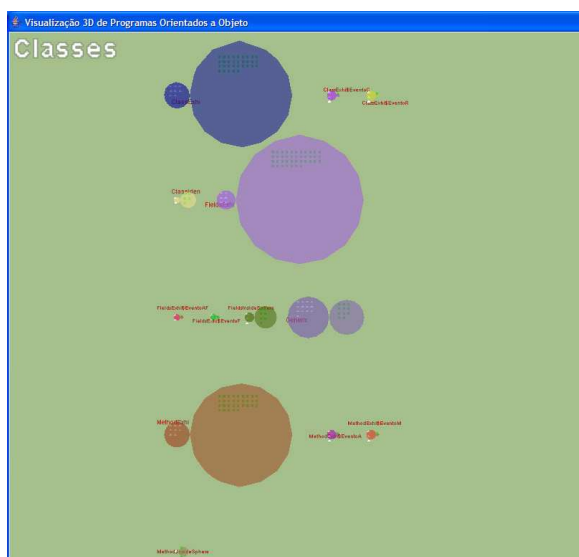


Figura 37: Aplicação de zoom na visualização das classes

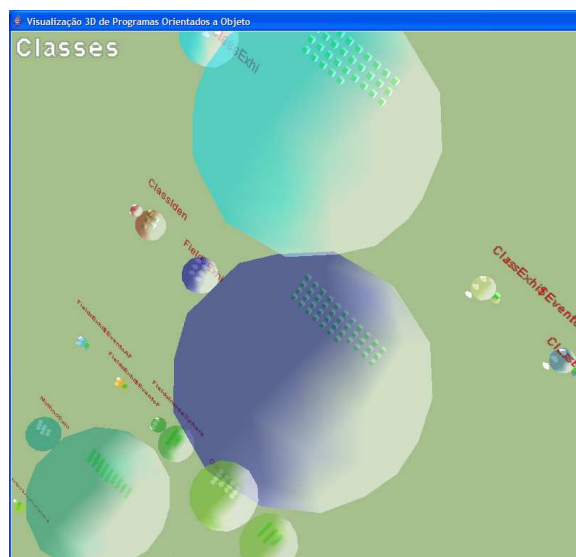


Figura 38: Aplicação de rotação na visualização das classes

4.4.1 Classe ClassIden.java

Esta classe, cujo único método é o `main`, é responsável pelo reconhecimento dos componentes de cada arquivo `.class` da estrutura de classes. Ao ser executada, a classe `ClassIden.java` realiza a importação dos pacotes descritos na Figura 39.

```
1 import org.apache.bcel.classfile.JavaClass;
2 import br.jabuti.lookup.Program;
3 import br.jabuti.lookup.RClassCode;
```

Figura 39: Pacotes importados para a classe `ClassIden.java`

O primeiro pacote faz parte da biblioteca *Byte Code Engineering Library* (BCEL). Esta biblioteca presta-se ao reconhecimento da estrutura de uma determinada classe. Os dois outros são componentes da biblioteca `br.jabuti.lookup` e também são usadas para a identificação dos componentes do programa, conforme demonstrado nas figuras seguintes.

A Figura 40 exhibe o código-fonte da classe `ClassIden.java`.

As linhas 1 e 2 declaram a classe `ClassIden` e o *array* de string `cls`. Em seguida, na linha 3, inicia-se o método `main`, único da classe. Os campos `cp` e `classeBase` são declarados nas linhas 4 e 5. Na linha seguinte, a classe `Program` é instanciada, utilizando os parâmetros especificados na subseção 4.1.2.

Na linha 7, é executado o método `getCodeClasses()`, da classe `Program`. Com isto, o *array* `cls` passa a armazenar os nomes das classes identificadas.

```
1 public class ClassIden{
2     static String[] cls;
3     public static void main(String[] args) throws Exception {
4         Program cp;
5         String classeBase;
6         cp = new Program(args[0], true, args[1], ".");
7         cls = cp.getCodeClasses();
8         RClassCode[] todasClasses = new RClassCode[cls.length];
9         JavaClass[] id = new JavaClass[cls.length];
10        classeBase = args[0];
11        for (int i = 0; i <= cls.length-1; i++) {
12            todasClasses[i] = (RClassCode) cp.get(cls[i]);
13            id[i] = todasClasses[i].getTheClass();
14        }
15        new ClassExhi(cls, id, classeBase, cp);
16    }
17 }
```

Figura 40: Código-fonte da classe `ClassIden.java`

Na linha 9 é criado o objeto `id`. Trata-se de um *array* que terá dimensão igual ao número de classes identificadas na estrutura, através do comando expresso na linha 7.

A linha 10 atribui à variável `classeBase` o primeiro argumento passado em linha de comando na chamada do programa. Trata-se da classe base ou principal.

Nas linhas 11 a 14 é usado um laço de repetição para que o *array* `id` contenha informações sobre as classes referenciadas pela classe base. Assim, neste ponto, `id` contém as informações de todas as classes que devem fazer parte da estrutura.

Finalmente, na linha 15 é chamada a classe `ClassExhi.class`, aquela que efetivamente exibe as classes. A partir desta classe, serão exibidos outros componentes do sistema: os métodos, seus campos e seu código-fonte. O primeiro argumento na chamada (`cls`) refere-se ao vetor contendo as classes que fazem parte da estrutura sob análise. O segundo argumento refere-se ao *array* `id`, que contém a identificação de todas as classes da estrutura. O terceiro argumento armazena o nome da classe base, primeiro argumento digitado na linha de execução do programa. Por fim, o argumento `cp` leva à classe `ClassExhi.java` o objeto da classe `Program`, que será utilizado para a avaliação das subclasses de uma determinada classe.

4.4.2 Classe `ClassExhi.java`

Esta classe exibe tridimensionalmente os componentes identificados como parte da estrutura de classes. Aqui, efetivamente, são utilizados comandos da API Java3D para a exibição da execução inicial contendo os pares de esferas. Na seqüência, o código desta classe será particionado em métodos para facilitar sua descrição.

Ao ser executada, esta classe importa diversas classes necessárias à execução de suas funcionalidades. A Figura 41 lista tais classes.

```
1 import java.awt.Font;
2 import java.awt.GraphicsConfiguration;
3 import java.awt.event.MouseAdapter;
4 import java.awt.event.MouseEvent;
.
.
22 import br.jabuti.lookup.Program;
23 import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
24 import com.sun.j3d.utils.behaviors.mouse.MouseTranslate;
25 import com.sun.j3d.utils.behaviors.mouse.MouseZoom;
26 import com.sun.j3d.utils.geometry.Sphere;
27 import com.sun.j3d.utils.geometry.Text2D;
28 import com.sun.j3d.utils.picking.PickCanvas;
29 import com.sun.j3d.utils.picking.PickResult;
30 import com.sun.j3d.utils.picking.PickTool;
31 import com.sun.j3d.utils.universe.SimpleUniverse;
```

Figura 41: Pacotes importados na classe `ClassExhi.java`

As classes importadas destinam-se à criação de interfaces com o usuário, criação e manipulação de objetos tridimensionais e reconhecimento dos componentes de um programa OO.

A Figura 42 exibe a parte seguinte do código-fonte da classe `ClassExhi.java`. Trata-se do construtor da classe, onde são instanciadas, entre outras, as classes `Method`, `Field` e `javaSuperClass`, responsáveis pela identificação dos métodos, atributos e superclasses das classes identificadas em `ClassIden.java`. Como a ferramenta deve exibir todos os métodos e atributos de todas as classes, os resultados são armazenados em *arrays*.

Nas linhas 12 a 18 são carregados tais vetores. As linhas seguintes tratam das chamadas dos métodos que irão possibilitar a exibição das classes, representadas por esferas. Os métodos `criaGrafoDeCena()` (linha 23) e `criaGrafoDeCenaH()` (linha 41) são aqueles que, de fato, escrevem os objetos em tela e serão tratados adiante nesta seção.

Ao término do construtor `ClassExhi`, inicia-se o método `exibeClasses()`, que implementa efetivamente a exibição da simbologia representativa das classes. É aplicada sobrecarga neste método para que seja exibida a esfera dos métodos ou a esfera dos atributos, dependendo do argumento passado na chamada a ele. A Figura 43 mostra a parte inicial do código do método que exibe a esferas dos métodos. Por sua similaridade funcional com este, o método `exibeClasses()`, responsável por exibir as esferas representativas dos atributos, não será abordado.

Conforme citado na seção 4.3, o tamanho das esferas varia conforme a quantidade de métodos e atributos das classes. A linha 5 cria uma instância da classe `Sphere`. A esfera

```

1 public ClassExhi(String[] clx, JavaClass[] javaClass, String argumento, Program cpC) {
2     generic = new Generic();
3     tG = new TransformGroup[clx.length];
4     t3D = new Transform3D[clx.length];
5     method = new Method[clx.length] [];
6     field = new Field[clx.length] [];
7     javaSuperClass = new JavaClass[clx.length] [];
8     this.argumento = argumento;
9     this.clx = clx;
10    this.javaClass = javaClass;
11    this.cpC = cpC;
12    for (int i = 0; i < method.length; i++)
13        method[i] = javaClass[i].getMethods();
14    for (int i = 0; i < field.length; i++)
15        field[i] = javaClass[i].getFields();
16    for (int i = 0; i < javaSuperClass.length;i++) {
17        javaSuperClass[i] = javaClass[i].getSuperClasses();
18    }
19    config = SimpleUniverse.getPreferredConfiguration();
20    canvas3D = new Canvas3D(config);
21    canvasSumario = new Canvas3D(config);
22    canvasArvore = new Canvas3D(config);
23    criaGrafoDeCena();
24    sumario = generic.criaSumario();
25    arvore = generic.criaArvore();
26    sumario.setCapability(BranchGroup.ALLOW_DETACH);
27    cena.setCapability(BranchGroup.ALLOW_DETACH);
28    arvore.setCapability(BranchGroup.ALLOW_DETACH);
29    cena.compile();
30    sumario.compile();
31    arvore.compile();
32    simpleU = new SimpleUniverse(canvas3D);
33    simpleSumario = new SimpleUniverse(canvasSumario);
34    simpleArvore = new SimpleUniverse(canvasArvore);
35    simpleU.getViewingPlatform().setNominalViewingTransform();
36    simpleU.addBranchGraph(cena);
37    simpleSumario.getViewingPlatform().setNominalViewingTransform();
38    simpleSumario.addBranchGraph(sumario);
39    simpleArvore.getViewingPlatform().setNominalViewingTransform();
40    simpleArvore.addBranchGraph(arvore);
41    criaGrafoDeCenaH();
42    simpleArvore.addBranchGraph(cenaH);
43    generic.exibeInstrucoes(config);
44 }

```

Figura 42: Construtor da classe `ClassExhi.java`

criada tem tamanho baseado na quantidade de métodos da classe. Na linha 10 é realizado o posicionamento da esfera, baseado em coordenadas que distribuem uniformemente os objetos em tela. As linhas 11 a 21 tratam de individualizar as esferas, de modo que possam ser escolhidas de forma individual para detalhamento e coloração. Das linhas 22 a 35 é especificada a aparência de cada esfera. Embora possuam cores distintas, todas são dotadas de transparência, para que se possa observar em seu interior os objetos representativos dos métodos e atributos.

Na Figura 44 é explicitada a segunda parte do método `exibeClasses()`. Aqui são criados os nomes das classes e colocados próximos às esferas (linhas 7 a 14). Na seqüência, são definidas as funcionalidades de translação e zoom das esferas. Da linha 31 em diante é chamado o método `MethodsInsideSphere()` e ajustadas coordenadas para exibição dos

```

1 public void exhibeClasses(String nomeC, Method[] metodosDaClasse, int
   contador, float posX, float posY, float corVerm, float corVerd, float
   corAzul) {
2     int qtdMetodos = metodosDaClasse.length;
3     totalMetodos += qtdMetodos;
4     float tamEsfera = (float) qtdMetodos;
5     esfera = new Sphere(tamEsfera*0.012f);
6     esfera.setCapability(Sphere.GENERATE_NORMALS);
7     esfera.setCapability(Sphere.ENABLE_GEOMETRY_PICKING);
8     esfera.setCapability(Sphere.ENABLE_APPEARANCE_MODIFY);
9     esfera.setUserData(nomeC);
10    Vector3f vectorEsf = new Vector3f(posX, posY, 0f);
11    t3D[contador] = new Transform3D();
12    tG[contador] = new TransformGroup();
13    tG[contador].setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
14    tG[contador].setCapability(Shape3D.ALLOW_APPEARANCE_READ);
15    tG[contador].setCapability(Shape3D.ENABLE_PICK_REPORTING);
16    tG[contador].setCapability(Shape3D.ALLOW_APPEARANCE_OVERRIDE_READ);
17    tG[contador].setCapability(Shape3D.ALLOW_APPEARANCE_OVERRIDE_WRITE);
18    t3D[contador].setTranslation(vectorEsf);
19    tG[contador].setTransform(t3D[contador]);
20    tG[contador].addChild(esfera);
21    tgPrinc.addChild(tG[contador]);
22    Appearance ap = new Appearance();
23    Material m = new Material();
24    m.setEmissiveColor(corVerm, corVerd, corAzul);
25    ap.setMaterial(m);
26    TransparencyAttributes tA = new TransparencyAttributes();
27    tA.setTransparency(0.4f);
28    tA.setTransparencyMode(TransparencyAttributes.NICEST);
29    ap.setTransparencyAttributes(tA);
30    Vector3f light1Direction = new Vector3f(-1f, 0.5f, 0f);
31    Color3f drColor = new Color3f(0.4f, 0.4f, 0.4f);
32    DirectionalLight dLgt = new DirectionalLight(drColor, light1Direction);
33    dLgt.setInfluencingBounds(new BoundingSphere());
34    esfera.setAppearance(ap);
35    cena.addChild(dLgt);

```

Figura 43: Primeira parte do método `exibeClasses()`, da classe `ClassExhi.java`

cubos dentro das esferas.

Na seqüência da classe, é implementado o método `criaGrafoDeCena()`, exibido na Figura 45. Este método faz o cálculo das coordenadas em que as esferas devem figurar antes de chamar o método `exibeClasses()`, visto anteriormente. As coordenadas iniciais são calculadas de acordo com o número de classes, objetivando o posicionamento das esferas tanto quanto possível na área central da tela.

O fragmento final do método, mostrado na Figura 46, destaca a criação aleatória das cores das esferas, a chamada do método de exibição e o cálculo das coordenadas da próxima esfera. É necessário salientar que as esferas ocuparão lugares definidos na tela e que as coordenadas são calculadas de modo a não permitir sobreposição entre uma esfera e outra. Conforme se extrai da linha 3, os cálculos das cores, coordenadas e chamadas dos métodos de exibição são efetuadas de acordo com o número de classes, através de um laço de repetição.

A linha 19 deste trecho mostra a chamada do método `addMouseListener(evt)`, re-

```

1     int tamFonte = 0;
2     float deslocTextY = 0f;
3     float deslocTextX = 0.06f;
4     if (nomeC.length() < 11) tamFonte = 12; else tamFonte = 10;
5     if (contador % 2 == 0) deslocTextY = 0.05f; else deslocTextY = -0.05f;
6     if (contador >= 4) deslocTextX = 0.17f;
7     TransformGroup textoC = new TransformGroup();
8     Transform3D transformText = new Transform3D();
9     Text2D text2D = new Text2D(nomeC, new Color3f(1.0f, 0.0f, 0.0f), "Arial", tamFonte, Font.PLAIN);
10    Vector3f vectorTextoC = new Vector3f(posX-deslocTextX, posY-deslocTextY, -0.09f);
11    transformText.setTranslation(vectorTextoC);
12    textoC.setTransform(transformText);
13    textoC.addChild(text2D);
14    tgPrinc.addChild(textoC);
15    textoC.setCapability(Shape3D.ALLOW_APPEARANCE_OVERRIDE_READ);
16    textoC.setCapability(Shape3D.ALLOW_APPEARANCE_OVERRIDE_WRITE);
17    BoundingSphere bounds = new BoundingSphere();
18    MouseZoom behaviorZoom = new MouseZoom();
19    behaviorZoom.setTransformGroup(textoC);
21    textoC.addChild(behaviorZoom);
22    behaviorZoom.setSchedulingBounds(bounds);
23    MouseTranslate behaviorTranslate = new MouseTranslate();
24    behaviorTranslate.setTransformGroup(textoC);
25    textoC.addChild(behaviorTranslate);
26    behaviorTranslate.setSchedulingBounds(bounds);
27    posX -= (tamEsfera * 0.012f) * 0.5f;
28    float posXIni = posX;
29    float posXMax = posX + (tamEsfera * 0.012f) * 0.75f;
30    posY += (tamEsfera * 0.012f) * 0.75f;
31    for (int j = 0; j < metodosDaClasse.length; j++) {
32        new MethodsInsideSphere(tgPrinc, cena, posX, posY, nomeC);
33        posX += 0.035f;
34        if (posX >= posXMax) {
35            posX = posXIni;
36            posY -= 0.035f;
37        }
38    }
39 }

```

Figura 44: Segunda parte do método `exibeClasses()`, da classe `ClassExhi.java`

```

1 public void criaGrafoDeCena() {
2     float pX;
3     float pY;
4     int i;
5     int numClasses = clx.length;
6     switch (numClasses) {
7         case 1: pX = -0.25f;
8                 pY = 0f;
9                 break;
10        case 2: pX = -0.5f;
11                pY = 0f;
12                break;
13        case 3: pX = -0.75f;
14                pY = 0f;
15                break;
16        default: pX = -0.75f;
17                pY = 0.55f;
18                break;
19    }

```

Figura 45: Início do método `criaGrafoDeCena()`, da classe `ClassExhi.java`

sponsável pelo reconhecimento do acionamento do mouse sobre uma das esferas.

Na seqüência da figura, a classe `EventoC` é implementada no mesmo arquivo que a


```

1      .
2      .
3      for (i = 0; i < clx.length; i++) {
4          .
5          .
6          Float verm = (float) Math.random();
7          Float verd = (float) Math.random();
8          Float azul = (float) Math.random();
9          exhibeClasses(clx[i],method[i],i,pX,pY,verm,verd,azul);
10         .
11         .
12         pX +=((desloca2*0.012f)+(desloca1*0.012f));
13         exhibeClasses(clx[i],field[i],i,pX,pY,verm,verd,azul);
14         pX +=((desloca2*0.012f)+(desloca1*0.012f))+0.2f;
15         if (pX >=0.75f) {
16             pX = -0.75f;
17             if (maiorM>=maiorF) pY -= (maiorM*0.012f)+0.4f; else pY -= (maiorF*0.012f)+0.4f;
18         }
19         canvas3D.addMouseListener(evt);
20     }

```

Figura 46: Parte final do método `criaGrafoDeCena()`, da classe `ClassExhi.java`

classe `ClassExhi`. Esta classe herda métodos e atributos da classe `MouseAdapter` e tem seus trechos mais relevantes retratados na Figura 47. Seu único método (`mouseClicked()`) é originário da classe `MouseAdapter` e é responsável por iniciar o canvas para o acionamento do mouse (linhas 4 a 7), capturar o acionamento (*pick*) e averiguar se o acionamento foi realizado numa área vazia ou numa esfera (linha 11). O acionamento do botão direito do mouse numa área vazia da tela não implica em qualquer ação. O mesmo efetuado numa esfera leva ao detalhamento dos métodos ou atributos da classe, dependendo da esfera clicada, conforme linhas 21 a 25.

O método `criaGrafoDeCenaH()` cria área na seção direita da tela (parte 2) que permite a reorganização das esferas e textos através do acionamento do mouse. Assim, após eventual aplicação de zoom, translação ou rotação nas esferas, este recurso permite a disposição das esferas em seu arranjo inicial.

Tal ação é efetivada pela classe `EventoR`, implementada de forma semelhante à classe `EventoC`. Em seu único método, a classe reconhece o acionamento do mouse, libera a cena atual das esferas e chama o método `criaGrafoDeCena()` para recriar o ambiente original. A classe `EventoR` tem suas principais linhas reproduzidas na Figura 48.

4.4.3 Classe `MethodExhi.java`

Esta classe é responsável por exibir, de forma detalhada, os métodos da classe selecionada inicialmente. A disposição dos cones nomeados segue padrão específico de distribuição em tela, cada um representando um método. De forma idêntica às esferas, é

```

1 private EventoC evt = new EventoC();
2 private class EventoC extends MouseAdapter {
3     public void mouseClicked( MouseEvent e ) {
4         PickCanvas pickCanvas = new PickCanvas (canvas3D, cena);
5         pickCanvas.setMode(PickTool.GEOMETRY_INTERSECT_INFO);
6         pickCanvas.setTolerance(2.0f);
7         pickCanvas.setShapeLocation(e);
8         PickResult results = null;
9         .
10        .
11        if (results != null){
12            Node n = results.getNode(PickResult.PRIMITIVE);
13            String nomeClasse = (String) n.getUserData();
14            iM = 0;
15            while (iM < javaClass.length){
16                if (javaClass[iM].getClassName().equals(nomeClasse) ||
17                    javaClass[iM].getClassName().concat(" ").equals(nomeClasse)) break;
18                iM++;
19            } //while
20            .
21            if (nomeClasse.endsWith(" ")){
22                new FieldsExhi(clx, field, javaSuperClass, simpleU, nomeClasse,
23                    iM, simpleSumario, canvas3D, canvasArvore, evt, evtR, simpleArvore, javaClass,cpC);
24            }
25            else
26                new MethodExhi(method, simpleU, javaSuperClass, nomeClasse,
27                    iM, simpleSumario, canvas3D, canvasArvore, evt, evtR, simpleArvore, javaClass,cpC);
28        } //if
29    } //mousePressed
30 } //EventoC

```

Figura 47: Classe EventoC()

```

1 private class EventoR extends MouseAdapter {
2     public void mouseClicked( MouseEvent e ) {
3         .
4         .
5         if (results == null){
6             cena.detach();
7             arvore.detach();
8             criaGrafoDeCena();
9             cena.setCapability(BranchGroup.ALLOW_DETACH);
10            cena.compile();
11            simpleU.getViewingPlatform().setNominalViewingTransform();
12            simpleU.addBranchGraph(cena);
13        }
14    }
15 }

```

Figura 48: Classe EventoR()

possível aplicar neste ambiente a rotação, translação e aplicação de zoom nos objetos, permitindo o aproveitamento das características de um ambiente tridimensional na visualização da informação.

Em muitos aspectos, a classe `MethodExhi.java` guarda semelhança conceitual com a `ClassExhi.java`, tratada anteriormente. O método construtor da classe, os métodos `exibeMetodos()`, `criaGrafoDeCenaM()` e a classe `EventoA` desempenham o mesmo papel que o método construtor, os métodos `exibeClasses()`, `criaGrafoDeCena()` e a classe `EventoC`, descritos na subseção 4.4.2. Ressalvadas diferenças básicas, tais como a troca

de esferas por cones, a exibição dos cones uma única vez (e não aos pares) e a apresentação da assinatura do método ao invés de ação de detalhamento no acionamento do mouse, a implementação destes componentes segue padrão idêntico à implementação dos componentes citados.

Merece destaque, no entanto, a implementação dos métodos `criaGrafoDeCenaH()` e `exibeHierarquia()`. O primeiro deles, cujo código completo é retratado na Figura 49, constrói o grafo de cena da árvore hierárquica correspondente à classe selecionada e realiza chamadas sucessivas do método `exibeHierarquia()`. As linhas 2 a 12 cuidam da preparação do ambiente para exibição da hierarquia, através da instanciação de classes e chamadas de métodos apropriados. A linha 13 ajusta as coordenadas iniciais da primeira esfera na árvore hierárquica. Esta primeira esfera, à propósito, será sempre a representação da classe `java.lang.Object`, superclasse de todas as classes Java.

```

1  public void criaGrafoDeCenaH() {
2      cenaH = new BranchGroup();
3      cenaH.setCapability(BranchGroup.ALLOW_DETACH);
4      generic.criaPlano(cenaH,0f,0.4f,0.2f);
5      tgPrincH = new TransformGroup();
6      tgPrincH.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
7      tgPrincH.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
8      cenaH.addChild(tgPrincH);
9      MouseRotate behaviorRotateH = new MouseRotate();
10     behaviorRotateH.setTransformGroup(tgPrincH);
11     tgPrincH.addChild(behaviorRotateH);
12     behaviorRotateH.setSchedulingBounds(new BoundingSphere());
13     float pX = 0.0f;
14     float pY = 1.4f;
15     for (int j = superClasses[iM].length-1; j >=0; j--) {
16         String nomeSuper = superClasses[iM][j].getClassName();
17         exibeHierarquia(pX, pY, nomeSuper, j);
18         pY -= 0.5f;
19     }
20     exibeHierarquia(pX,pY,nomeClass,superClasses[iM].length-1);
21     subclassesM = cpM.getSubClassClosure(nomeClass);
22     if (subclassesM.length==1) {
23         pY -= 0.5f;
24         exibeHierarquia(pX, pY, subclassesM[0],1);
25     }
26     else {
27         for (int j = 0; j < subclassesM.length; j++) {
28             pX -= 0.45f;
29             pY -= 0.5f;
30             exibeHierarquia(pX, pY, subclassesM[j],j);
31             pY += 0.5f;
32             pX += 1.35f;
33         }
34     }
35     canvasArvore.addMouseListener(evtA);
36 }

```

Figura 49: Método `criaGrafoDeCenaH()`, da classe `MethodExhi.java`

Nas linhas 15 a 19 são avaliadas as superclasses da classe selecionada e chamado o método de exibição das esferas que compõem a árvore hierárquica. Os argumentos `pX`, `pY`, `nomeSuper` e `j` são, respectivamente, a coordenada X da esfera, sua coordenada Y,

o nome da superclasse e seu índice no vetor de superclasses. Na linha 20, o método de exibição das esferas da hierarquia é chamado novamente, desta vez para que seja colocada na árvore a esfera correspondente à própria classe sob análise.

Na linha 21 são avaliadas as subclasses da classe selecionada. A partir da linha 22 até a 34, o processo de chamada de exibição se repete. Entretanto, por haver possibilidade de existir mais de uma classe filha para uma classe pai, as coordenadas são calculadas de modo a deixar as filhas alinhadas, como mostra a Figura 50.



Figura 50: Árvore hierárquica da classe `ClassExhi.java`

No âmbito da árvore hierárquica, o acionamento do botão direito do mouse nas esferas leva ao detalhamento dos métodos ou atributos desta classe, dependendo da escolha preliminar do usuário. Vale salientar que o acionamento do mouse na esfera `java.lang.object` leva à exibição inicial das classes e que o acionamento do mouse na própria classe clicada não provoca efeito no programa.

4.4.4 Classe `FieldsExhi.java`

Esta classe torna possível a visualização dos atributos de uma classe de forma detalhada. Do mesmo modo que ocorre na classe `MethodExhi.java`, os objetos (cubos) são dispostos uniformemente na tela, nomeados e aptos a receberem rotação, translação e aplicação de zoom.

Há que se destacar aqui a diferenciação de cor aplicada aos cubos de acordo com cada tipo de modificador de acesso do atributo. Conforme mostra o trecho da Figura 51, cada modificador corresponde a uma coloração diferente aplicada ao método `criaAparencia()`.

```

1 public void exhibeCampos(float randomX, float randomY, String nomeC,
2                         String tipoF, int modificador) {
3     .
4     .
5     switch (modificador) {
6         case 1: box = new Box (0.07f,0.07f,0.07f,criaAparencia(1f,0f,0f));
7             break;
8         case 2: box = new Box (0.07f,0.07f,0.07f,criaAparencia(0f,1f,0f));
9             break;
10        case 3: box = new Box (0.07f,0.07f,0.07f,criaAparencia(0f,0f,1f));
11            break;
12        default: box = new Box (0.07f,0.07f,0.07f,criaAparencia(0.6f,0.6f,0.6f));
13            break;
14    }
15    .
16    .
17 }

```

Figura 51: Método `exibCampos()`, da classe `FieldsExhi.java`

O parâmetro `modificador` é oriundo do método `criaGrafoDeCenaF()`, que avalia o modificador de acesso de cada atributo da classe selecionada, de acordo com a Figura 52. O modificador `Private` é associado ao inteiro 1. O modificador `Public` ao inteiro 2 e o modificador `Static` ao inteiro 3. Não ocorrendo nenhuma sentença verdadeira nas linhas de 7 a 13, o campo será considerado como pertencente a categoria `Package`, significando que serão visíveis para todas as classe de um mesmo pacote (SANTOS, 2003).

```

1 public void criaGrafoDeCenaF() {
2     .
3     .
4     for (int d=0;d<clxF.length;d++)
5         if(tipo.toString().equals(clxF[d].trim()))
6             tipoAux = tipo.toString();
7     if (campos[iF][j].isPrivate()==true) acesso = 1; //Private
8     else
9         if (campos[iF][j].isPublic()==true) acesso = 2; //Public
10        else
11            if (campos[iF][j].isStatic()==true) acesso = 3; //Static
12            else acesso = 0;
13        exhibeCampos(pX, pY, nomeCampo,tipoAux,acesso);
14        pX += 0.27f;
15        if (pX >=0.8f) { //salto de linha
16            pX = -0.85f;
17            pY -= 0.24f;
18        }
19    }
20 }

```

Figura 52: Método `criaGrafoDeCenaF()`, da classe `FieldsExhi.java`

Após a avaliação do modificador, o método de exibição dos campos é chamado na linha 14. Logo em seguida, são reajustadas as coordenadas para exibição do próximo cubo.

4.4.5 Classe `Generic.java`

Esta classe contém métodos que cuidam de aspectos subsidiários da ferramenta, tais como criação das áreas de legenda (aqui chamadas genericamente de sumário), criação de plano sobre o qual os objetos são colocados, a criação de janela das instruções de uso da ferramenta e, principalmente, as janelas em que as diversas seções da ferramenta são postadas.

4.4.6 Classes `FieldsInsideSphere.java` e `MethodsInsideSphere.java`

Estas classes tratam da inserção de cones e cubos dentro das esferas de métodos e atributos, respectivamente. Conforme tratado, tais esferas são exibidas na execução inicial da ferramenta e são dotadas de transparência para que os objetos nela contidos possam ser vistos pelo observador.

A codificação destas classes segue os padrões vistos até aqui, ou seja, a definição de coordenadas e a chamada de um método de criação e exibição do objeto.

O presente capítulo detalhou aspectos operacionais e de implementação da ferramenta de visualização de software. As tecnologias envolvidas na ferramenta foram abordadas, bem como as metáforas utilizadas na representação dos componentes de um programa. A seguir, serão apresentados resultados e discussões acerca da submissão de dois programas à análise da ferramenta.

5 *Resultados e discussões*

A proposta principal da ferramenta é a exibição de todas as classes, métodos e atributos do programa, oferecendo ao usuário a possibilidade de interagir com estes componentes de modo a obter detalhes sobre eles. Além disso, faz parte de seu objetivo a exibição da hierarquia entre todas as classes que fazem parte da estrutura do programa, de modo a transmitir ao usuário o grau de relacionamento entre elas.

Os exemplos de execução fornecidos até o momento fazem referência à análise do programa Java desenvolvido para a própria construção da ferramenta. Embora tal procedimento tenha sua devida importância, a submissão de um só sistema à ferramenta não é capaz de contemplar os diversos perfis que um programa Java pode apresentar. Por se tratar de instrumento genérico, a ferramenta deve estar apta a exibir a estrutura de programas Java complexos, levando em conta a quantidade de classes, métodos e atributos que o compõem.

Assim, o presente capítulo destina-se a apresentar casos de submissão de dois programas Java à ferramenta desenvolvida para os fins deste projeto e, neste contexto, são comentadas as características da ferramenta.

Para efeitos de comparação entre a visualização proporcionada pela ferramenta e uma visualização 2D de um programa, será apresentado o diagrama de classes de um dos programas analisados.

5.1 Programa `Editor.java`

`Editor.java` (PAWLAN, 2001) é um programa que objetiva explicar como usar o método `InvokeLater` em aplicações *Multithreaded*. Seu código possui 15 classes, 60 métodos e 31 atributos e, embora seu tamanho não seja de grandes dimensões, apresenta a característica de possuir mais classes hierarquizadas do que o exemplo anterior.

A execução inicial da ferramenta aplicada a este programa encontra-se na Figura 53.

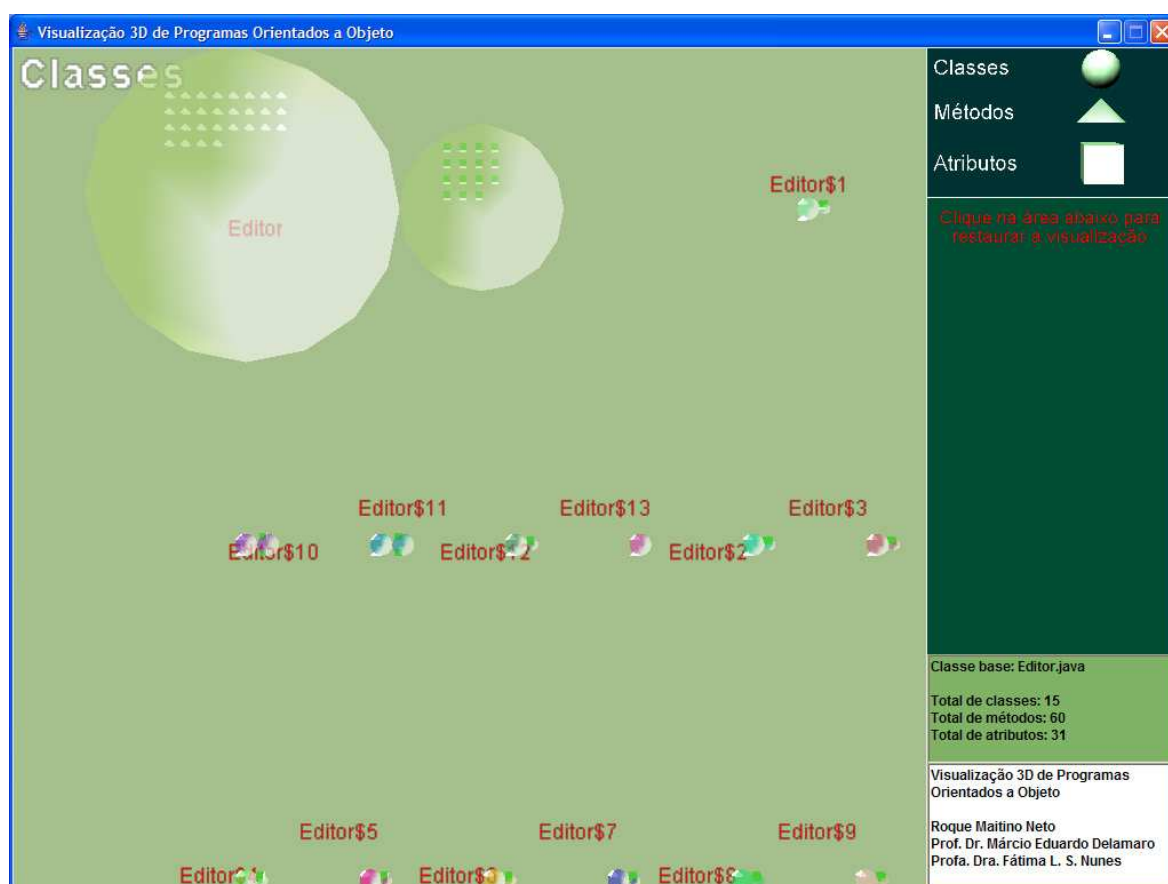


Figura 53: Execução inicial da ferramenta em análise ao programa *Editor.java*

A Figura 54 mostra o detalhamento dos métodos da classe *Editor.java*. Há que se salientar a grande quantidade de classes em nível superior a esta, destacado na parte 2 da imagem. Para que a árvore hierárquica pudesse ter visualização completa, foi necessária a aplicação de zoom. Caso houvesse necessidade, poder-se-ia aplicar os recursos de rotação e translação também neste caso.

Conforme se depreende pela observação do tamanho das demais esferas na visualização inicial, as outras classes possuem poucos métodos e atributos. No caso da classe *Editor\$13*, por exemplo, sequer há atributos e, portanto, a esfera correspondente é omitida da visualização.

Levados estes fatores em consideração, a ferramenta mostrou-se adequada à visualização de programa com o perfil apresentado. Tanto na apresentação dos componentes do programa, quanto na apresentação da hierarquia entre classes, o uso dos recursos próprios do ambiente tridimensional teve a sua utilidade constatada.

No caso específico da análise da hierarquia entre as classes, uma limitação própria da ferramenta pôde ser constatada: a impossibilidade de se saber, de antemão, quantas e



Figura 54: Detalhamento dos métodos da classe *Editor.java*, com aplicação de zoom na parte 2

quais esferas eventualmente possam estar fora do campo de visualização do usuário assim que a imagem é exibida.

Uma solução para este caso seria a limitação no número de níveis de hierarquia entre as classes. Assim, as esferas ocupariam apenas o espaço a elas reservado na porção da imagem em que é exibida a hierarquia. Tal recurso, no entanto, deixaria eventualmente de fora da observação de uma ou mais classes, o que seria contrário à proposta da ferramenta.

Outra solução viável seria a exibição da estrutura hierárquica em menor escala quando o número de objetos a serem exibidos ultrapassasse certa quantidade. Esta providência será rediscutida no Capítulo 6, como parte das propostas de implementação futura.

5.2 Programa *JavaKitTest.java*

Este programa foi construído para criar um editor de código-fonte, com base em implementação customizada da classe *EditorKit*, usada para adicionar cores à classe *JEditorPane*. O editor é projetado para editar código-fonte Java (PRINZING, 2006).

A Figura 55 mostra o diagrama de classes do programa.

Antes da análise da submissão do programa *JavaKitTest.java* à ferramenta implementada para este projeto, faz-se necessário ressaltar a diferença entre a visualização das classes (com seus métodos e atributos) através do Diagrama de Classes e através da fer-

ramenta desenvolvida. No que diz respeito a qualidade da visualização, a imagem do Diagrama mostra-se pouco apropriada para impressão em uma lauda. Como não existem recursos para transladar ou aplicar zoom na imagem, a leitura do diagrama tende a ficar prejudicada na medida que o número de componentes aumenta.

Por outro lado, a visualização do mesmo contexto em ambiente tridimensional através da ferramenta oferece, além da possibilidade de exploração das informações em níveis, uma disposição otimizada dos componentes em tela, o que facilita a análise visual dos mesmos.

A escolha desta classe apóia-se no fato do programa possuir um número quantitativamente apropriado de componentes para análise de resultados: 20 classes, 53 métodos e 166 atributos. A Figura 56 mostra a aplicação da ferramenta no programa.



Figura 56: Execução inicial da ferramenta aplicada ao programa *JavaKitTest.java*

Por meio dela, é possível observar que na visualização inicial, apenas 10 das 20 classes são visíveis. Através do recurso da translação, ativado pelo botão direito do mouse, é possível a visualização das demais classes, conforme mostra a Figura 57.

Havendo necessidade da visualização simultânea das 20 classes, deve-se usar o recurso da aplicação de zoom, através do botão central do mouse. Além disso, destaca-se que a classe *Token.java* possui um número elevado de atributos, o que torna a esfera cor-

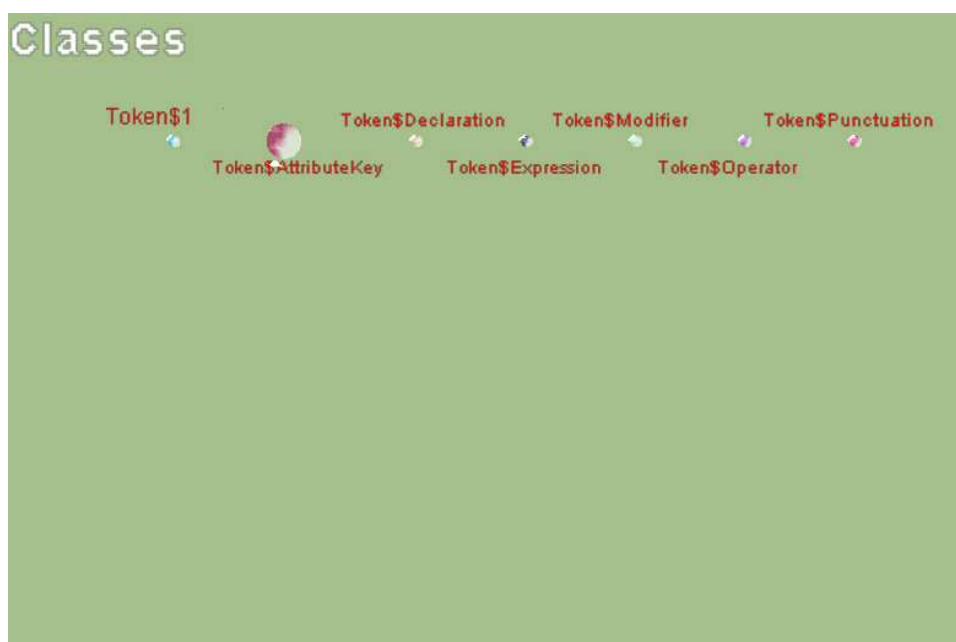


Figura 57: Demais classes de `JavaKitTest.java`

respondente bastante grande. Mais uma vez, a aplicação dos recursos próprios de um ambiente tridimensional da ferramenta sanam a falta de uma imediata e total visualização dos objetos em cena. Observa-se ainda que, em razão de seu grande tamanho, a esfera de atributos da classe `Token.java` fica sobreposta à esfera de atributos da classe `JavaDocument$DocumentInputStream.java`. A rotação da cena para a esquerda coloca a esfera sobreposta em posição acessível ao usuário, podendo, então, ser acionada, conforme mostra a Figura 58.

Embora outros tantos programas tenham sido cogitados para incrementar esta análise, foram observadas variações de pouca significância nas visualizações para efeitos de apuração de resultados.

5.3 Avaliação dos resultados

Além das evidentes virtudes da ferramenta, os programas escolhidos também foram capazes de expor certas limitações em sua implementação. Exemplo disto é a impossibilidade de observação dos nomes relacionados aos objetos após a aplicação de rotação nestes objetos. Tal limitação advém, em parte, da falta de recurso conhecido na API Java3D que permita a recolocação de objetos (texto, por exemplo) em tempo de execução.

Outro aspecto que deve ser ressaltado neste sentido é o crescimento excessivo das esferas de métodos e/ou atributos quando tais componentes se apresentam em grande

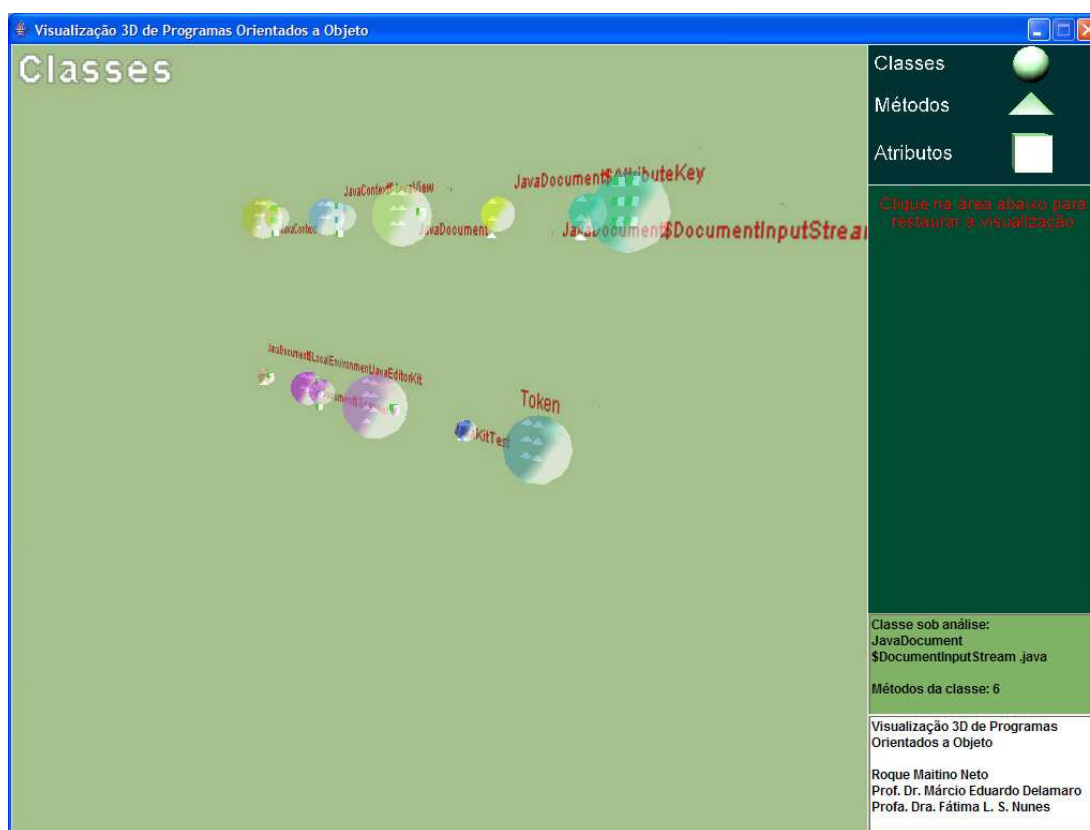


Figura 58: Rotação aplicada a visualização do programa `JavaKitTest.java`

número nas classes analisadas. Embora o recurso do zoom possa ser usado para diminuir o tamanho dos objetos em tela, este afastamento no eixo z para a observação do objeto de tamanho maior pode prejudicar a observação dos objetos menores.

Não obstante a isto, a ferramenta trata com sucesso dos diversos perfis de programas testados, oferecendo boa visualização de seus componentes. Uma simples comparação entre as figuras apresentadas como exemplos de execução da ferramenta e o diagrama de classes do programa apresentado na Figura 24, no Capítulo 4, é capaz de expor as diferentes visualizações de uma mesma realidade.

Em programas que apresentem um número considerável de classes (mais que 20, por exemplo), a visualização oferecida pelo diagrama de classes pode dificultar a distinção dos atributos e métodos de cada classe. Ao contrário, a visualização tridimensional pode oferecer a observação de contextos particulares ou gerais, conforme a necessidade do usuário. Além disso, a possibilidade de interação, a possibilidade de utilização de cores e texturas diferentes para componentes diferentes e a visualização em níveis proporcionam vantagens para a visualização tridimensional. Pelos exemplos apresentados, verifica-se que a ferramenta oferece tais funcionalidades como parte de sua construção.

Conforme visto na seção 2.2, Chittaro (2001) fornece os objetivos gerais das técnicas de exibição de dados, citando, entre outros, a possibilidade de exploração dos dados em vários níveis de abstração. A ferramenta cumpre tal objetivo ao possibilitar, através do uso do mouse, que o usuário transfira a cena de um ambiente genérico (visualização de classes) para um ambiente particularizado, onde os métodos ou atributos da classe são visíveis em detalhes.

Outro objetivo citado por Chittaro (2001) trata da descoberta de detalhes e relações entre certas entidades, tarefa que poderia ser árdua sem o recurso da visualização. Quando exhibe a disposição hierárquica da classe sob análise de maneira simples e com compreensão imediata, a ferramenta pretende transmitir ao observador a idéia de como ela se relaciona com outras classes. Este relacionamento pode se dar através do fornecimento ou do uso de funcionalidades de uma classe em relação a outra(s).

Por fim, destaca o citado autor que também deve ser objetivo das técnicas de exibição a compreensão mais profunda dos dados exibidos. Neste sentido, a ferramenta apresenta os atributos das classes em cores que distinguem os vários modificadores de acesso aplicáveis a um atributo. Um objeto que representa um atributo que compõe agregação com uma das classes da estrutura sob análise é apresentado igualmente de forma distinta. Desta forma, o observador poderá, na mesma cena, diferenciar diversos tipos de atributos de forma imediata, obtendo, assim, um entendimento maior de tais componentes.

A seção 2.3 destaca aspectos relevantes de uma ferramenta de visualização. Algumas destas funcionalidades destacadas por Maletic, Leigh e Marcus (2001a) são contempladas pela ferramenta, que oferece (i) visão geral dos dados apurados, disponível assim que a ferramenta é executada; (ii) possibilidade de aplicação de zoom em determinado conjunto de objetos, bem como a aplicação de rotação e translação nestes objetos. Com o acionamento do mouse, é possível ainda recuperar a visualização contextual dos dados; (iii) detalhes de itens da visualização ao comando do usuário e (iv) possibilidade de visualização de relações entre itens da visualização, especificamente entre classes.

A seção 2.4 apresenta várias dimensões para a visualização de software. Price, Baecker e Small (1998) destacam, entre outras, o meio de exibição como uma das condições de existência de um programa de visualização de programas.

Neste sentido, a ferramenta em questão vale-se do uso dos atuais monitores de vídeo para viabilizar o uso de ambiente tridimensional na exibição dos componentes de um programa.

Este capítulo apresentou a aplicação da ferramenta em dois programas Java cujos perfis diferiam entre si e daquele implementado para a própria construção da ferramenta. Resultados obtidos com base nestas aplicações foram discutidos e algumas limitações foram trazidas à tona. O capítulo seguinte traz considerações finais do autor sobre a importância do trabalho e relaciona possibilidades de continuidade de desenvolvimento futuro da ferramenta.

6 *Considerações Finais*

A visualização da informação, tratada de forma geral, estende seus benefícios por variadas e numerosas áreas da atividade humana. A própria informática, reconhecida como criadora e propulsora desta tendência, beneficia-se das vantagens da observação simbólica e organizada de determinado conjunto de informações. Sem dúvida, a visualização de programas é parte importante deste contexto.

Neste projeto, após pesquisa envolvendo visualização de software e assuntos relacionados, foi desenvolvida uma ferramenta de visualização de programas escritos em linguagem Java. Utilizando-se recursos desenvolvidos por terceiros e implementação própria, a ferramenta é capaz de transformar elementos da linguagem Java em objetos tridimensionais.

Tal desenvolvimento trouxe à tona desafios próprios, tais como a escolha das metáforas adequadas e a seleção dos elementos (e suas relações) da linguagem a serem representados, além dos naturais obstáculos na criação de programas em ambientes até então pouco familiares ao programador.

A escolha das formas de representação dos objetos (metáforas) deu-se por conta da simplicidade de sua implementação aliada a sua capacidade de transmitir o comportamento dos componentes do programa sob análise. Tais componentes foram selecionados de forma a proporcionar ao usuário uma visão ampla do programa, desde a primeira imagem de execução.

Embora a visualização de um programa orientado a objetos tenha também outros propósitos, há que se destacar a função pedagógica de uma representação desta natureza. É de se imaginar que a aprendizagem de conceitos relativos ao paradigma seja mais efetiva através da sua percepção visual. A possibilidade de visualização da hierarquia entre classes – outra característica da ferramenta – pode auxiliar no entendimento de como determinados recursos de uma classe podem ser aproveitados em outras de hierarquia inferior.

Após a submissão de alguns programas com perfis diversos à ferramenta, foi possível

constatar sua utilidade em visualizações de quantidades reduzidas de objetos até visualizações de várias classes, com concentrações de mais de 80 atributos em algumas delas. Dois destes programas foram tomados como estudos de caso e apresentados no Capítulo 5.

Reunidos tais fatos, é presumível a conclusão de que, embora apresente limitações de visualização em grande escala, a ferramenta desenvolvida atinge os objetivos aos quais se propôs em sua concepção. Algumas das possíveis e naturais evoluções são a inclusão da análise e exibição de pacotes, a inclusão da visualização das interfaces implementadas pelas classes, a exibição do código-fonte dos métodos e/ou classes e a possibilidade de escolha das metáforas por parte do usuário.

Além destas melhorias, a visualização dos objetos em escala se constitui em necessidade de implementação futura. Dependendo do número de objetos a serem exibidos, a ferramenta deve adequar a escala destes objetos de maneira a mostrá-los de uma só vez, deixando ao usuário a possibilidade de alterar a escala através da aplicação de zoom.

Tais implementações acarretariam, sem dúvida, um sentido ainda maior de completude ao trabalho, tornando-a mais apta à exibição de sistemas de grandes proporções.

Por fim, a possibilidade da inclusão do presente trabalho como parte integrante da ferramenta JaBUTi. Desenvolvida por Vincenzi et al. (2003), esta ferramenta é projetada para realizar teste estrutural em programas orientados a objeto sem a necessidade da presença de seu código fonte. Assim, este trabalho serviria como forma de visualizar tridimensionalmente o programa a ser testado.

Referências

- ADMS, L. *Visualização e Realidade Virtual*. São Paulo: Makron Books, 1994.
- ALFERT, K.; ENGELN, F. Experiences in 3-dimensional visualization of java class relations. *Transactions of the SDPS*, v. 5, n. 3, p. 91–106, September 2001.
- APACHE. *BCEL*. Forest Hill, MD, USA, 2002–2003. Disponível em: <<http://jakarta.apache.org/bcel/index.html>>. Acesso em: 1 fev. 2006.
- BALINT, I. Z. *Introduction to 3D*. [S.l.]: Media, 2001.
- BALZER, M.; NOACK, A.; DEUSSEN, O. Software landscapes: Visualizing the structure of large software system. In: VISSYM (Ed.). Germany: Eurographics Association, 2004. p. 261–266.
- BOUVIER, D. J. *Getting Started with the Java 3D API*. 1.5.1. ed. Mountain View, California, 2000.
- BREWER, I.; MACEACHREN, A. M.; ABDO, H. Collaborative geographic visualization: Enabling shared understanding of environmental processes. In: *INFOVIS '00: Proceedings of the IEEE Symposium on Information Visualization 2000*. Washington, DC, USA: IEEE Computer Society, 2000. p. 137. ISBN 0-7695-0804-9.
- CARD, S.; MACKINLAY, J.; SHNEIDERMAN, B. Readings in information visualization using vision to think. In: . San Francisco, CA: Morgan Kaufmann, 1999.
- CHITTARO, L. Information visualization and its application to medicine. *Artificial Intelligence in Medicine*, Udine, v. 22, n. 2, p. 81–88, 2001.
- COUSINS, S.; KAHN, M. *The Visual Display of Temporal Information*. [S.l.], 1991.
- DEITEL, H. M.; DEITEL, P. J. *Java: How to program*. 4. ed. New Jersey: Prentice Hall, 2002. 1500 p.
- DELAMARO, M. E. *Automatização do Teste Estrutural de Agentes Móveis*. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação (ICMC-USP), São Carlos, 2005.
- DERSHEM, H. L.; VANDERHYDE, J. Java class visualization for teaching object-oriented concepts. In: *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*. Atlanta, Georgia, United States: ACM Press, 1998. p. 53–57. ISBN 0-89791-994-7.
- EDELSON, D. C.; PEA, R. D.; CLARK, B. A. The sscivee project - supportive scientific visualization environments for education. 1999.

- EISENSTADT, M.; PRICE, B.; DOMINGUE, J. Software visualization as a pedagogical tool: Redressing some its fallacies. Ablex, p. 311–339, 1999.
- FAST, K. Information visualization: Failed experiment or future revolution? In: *5th Annual Information Architecture Summit*. Austin, Texas: [s.n.], 2004.
- FREITAS, C. M.; LUZZARDI, P. R.; CAVA, R. A. On evaluating information visualization techniques. In: . [s.n.], 2002. Disponível em: <citeseer.ist.psu.edu/freitas02evaluating.html>.
- GALLAGHER, R. S. *Computer Visualization: Graphics techniques for scientific and engineering analysis*. New York: CRC Press, 1994. ISBN 0849390508.
- GEISLER, G. Making information more accessible: A survey of information visualization applications and techniques. 1998. Disponível em: <<http://www.ils.unc.edu/geisg/info/infovis/paper.html>>.
- GERSHON, N.; EICK, S. G. Guest editor's introduction: Information visualization. *IEEE Computer Graphics and Applications*, v. 17, n. 4, p. 29–31, 1997.
- GORDIN, D. N.; EDELSON, D. C.; GOMEZ, L. M. Scientific visualization as an interpretive and expressive medium. In: *Proceedings of the Second International Conference on the Learning Sciences*. Evanston, IL: [s.n.], 1996. p. 409–414.
- GOSLING, J.; JOY, B.; STEELE, G. *The Java Language Specification*. 3. ed. Boston: Addison-Wesley, 2005. (The Java Series).
- GREEN, R. *Java Glossary*. Canada, 2006. Disponível em: <<http://mindprod.com/jgloss/javaclass.html>>. Acesso em: 2 fev. 2006.
- HEARN, D.; BAKER, P. M. *Computer Graphics: C version*. 2. ed. New Jersey: Prentice Hall, 1997.
- HORSTMANN, C. S.; CORNELL, G. *Core Java 2: Fundamentos*. São Paulo: Makron Books, 2001.
- HOUAISS, A.; VILLAR, M. de S. *Dicionário da Língua Portuguesa*. 1. ed. Rio de Janeiro: Objetiva, 2001.
- KNIGHT, C.; MUNRO, M. C. Virtual but visible software. In: *Proceedings of the International Conference on Information Visualisation*. [S.l.]: IEEE Computer Society, 2000. p. 198. ISBN 0-7695-0743-3.
- LINDHOLM, T.; YELLIN, F. *The Java Virtual Machine Specification*. 2. ed. California, US: [s.n.], 1996.
- MACEACHREN, A. M.; BOSCOE, F. P.; HAUG, D. Geographic visualization: Designing manipulable maps for exploring temporally varying georeferenced statistics. In: *INFOVIS '98: Proceedings of the 1998 IEEE Symposium on Information Visualization*. Washington, DC, USA: IEEE Computer Society, 1998. p. 87. ISBN 0-8186-9093-3.
- MACKINLAY, J. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, ACM Press, v. 5, n. 2, p. 110–141, April 1986.

- MALETIC, J.; MARCUS, A.; COLLARD, M. A task oriented view of software visualization. In: *In Proceedings of the 1st IEEE Workshop on Visualizing Software for Understanding and Analysis - VISSOFT 2002*. [S.l.: s.n.], 2002.
- MALETIC, J. I.; LEIGH, J.; MARCUS, A. Visualizing object-oriented software in virtual reality. In: *IWPC '01: Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*. [S.l.]: IEEE Computer Society, 2001. p. 26.
- MALETIC, J. I.; LEIGH, J.; MARCUS, A. Visualizing software in an immersive virtual reality environment. In: *Proceedings of ICSE'01 Workshop on Software Visualization*. Toronto, Canada: [s.n.], 2001. p. 49–54.
- MALLOY, B. A.; POWER, J. F. Using a molecular metaphor to facilitate comprehension of 3d object diagrams. In: *Proceedings of 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Dallas, Texas: IEEE Computer Society, 2005. p. 8.
- MARCUS, A.; FENG, L.; MALETIC, J. I. 3d representations for software visualization. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York: ACM Press, 2003. p. 27–36. ISBN 1-58113-642-0.
- MICROSOFT. *Essentials of Object-Oriented Programming*. Redmond, WA, 2001.
- MICROSYSTEMS, S. *The Java 3D API Specification*. Santa Clara, CA, jun 2002.
- MICROSYSTEMS, S. Java the source for developers. In: *Java the source for developers*. [s.n.], 2005. Disponível em: <<http://java.sun.com/products/java-media/3D/java3d-features.html>>. Acesso em: 16 mar. 2005.
- MUNZNER, T. Guest editor's introduction: Information visualization. *IEEE Computer Graphics and Applications*, v. 22, n. 1, p. 20–21, 2002.
- NASCIMENTO, H. A. D. do; FERREIRA, C. B. R. Visualização de informações: Uma abordagem prática. *Sociedade Brasileira de Computação*, p. 1265 – 1312, julho 2004.
- NIELSEN, F. A.; HANSEN, L. K. Interactive information visualization in neuroimaging. In: *Workshop on New Paradigms in Information Visualization and Manipulation*. [S.l.: s.n.], 1997. p. 62–65.
- NORTH, C.; KORN, F. Browsing anatomical image databases: a case study of the visible human. In: *CHI '96: Conference companion on Human factors in computing systems*. New York, NY, USA: ACM Press, 1996. p. 414–415. ISBN 0-89791-832-0.
- PANAS, T.; BERRIGAN, R.; GRUNDY, J. A 3d metaphor for software production visualization. In: *IV '03: Proceedings of the Seventh International Conference on Information Visualization*. [S.l.]: IEEE Computer Society, 2003. p. 314. ISBN 0-7695-1988-1.
- PAWLAN, M. Multithreaded swing applications. september 2001. Disponível em: <<http://java.sun.com/developer/technicalArticles/Threads/swing/>>.
- PLAISANT, C.; MUSHLIN, R.; SNYDER, A. *LifeLines: Using Visualization to Enhance Navigation and Analysis of Patient Records*. [S.l.], 1998. Disponível em: <citeseer.ist.psu.edu/plaisant98lifelines.html>.

- PRICE, B.; BAECKER, R.; SMALL, I. An introduction to software visualization. In: *Software Visualization*, Stasko, J., Dominique, J., Brown, M., and Price, B. London: MIT Press, 1998. p. 4–26.
- PRINZING, T. Customizing a text editor. 2006. Disponível em: <http://java.sun.com/products/jfc/tsc/articles/text/editor_kit/index.html>. Acesso em: 31 jul. 2006.
- RUSS, J. C. *The Image Processing Handbook*. 4. ed. New York: CRC Press, 2002.
- SANTOS, R. *Introdução à programação orientada a objetos usando Java*. [S.l.]: Editora Campus, 2003. 319 p.
- SOWIZRAL, H.; RUSHFORTH, K.; DEERING, M. *The Java 3D API Specification*. California, US, 2000. Disponível em: <http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1.2_API/j3dguide>. Acesso em: 30 jan. 2006.
- STASKO, J. T. Three-dimensional computation visualization. In: *Proc. IEEE Symp. Visual Languages, VL*. [S.l.: s.n.], 1993.
- VINCENZI, A. M. R. et al. *JaBUTi - Java Bytecode Understanding and Testing*. São Carlos, 2003.
- WARE, C. *Information Visualization: Perception for design*. 2. ed. San Francisco, CA: Morgan Kaufmann Publishers, 2004.
- WARE, C.; HUI, D.; FRANCK, G. Visualizing object oriented software in three dimensions. In: *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*. Toronto: IBM Press, 1993. p. 612–620.
- WHATIS.COM. Visualization. In: *Whatis.com, the leading IT encyclopedia and learning center*. [s.n.], 2005. Disponível em: <http://whatis.techtarget.com/definition/0,,sid9_gci213311,00.html>. Acesso em: 20 jan. 2005.
- WIKIPEDIA. Metaphor. In: *Wikipedia, the free encyclopedia*. [s.n.], 2005. Disponível em: <<http://en.wikipedia.org/wiki/Metaphor>>. Acesso em: 25 mar. 2005.
- WIKIPEDIA. Scientific visualization. In: *Wikipedia, the free encyclopedia*. [s.n.], 2005. Disponível em: <http://en.wikipedia.org/wiki/Scientific_visualization>. Acesso em: 20 jan. 2005.
- WIKIPEDIA. Visualization. In: *Wikipedia, the free encyclopedia*. [s.n.], 2005. Disponível em: <<http://en.wikipedia.org/wiki/Visualization>>. Acesso em: 20 jan. 2005.
- WILSON, B. S. *Object-Oriented Programming Principles*. 1.1. ed. Rochester, NY, 2000.
- WISS, U.; CARR, D.; JONSSON, H. Evaluating three-dimensional information visualization designs: A case study of three designs. In: *IV '98: Proceedings of the International Conference on Information Visualisation*. [S.l.]: IEEE Computer Society, 1998. p. 137. ISBN 0-8186-8509-3.
- YATES, R. B.; RIBEIRO, B. *Modern Information Retrieval*. New York: Addison-Wesley, 1999.