UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SAIRO RAONÍ DOS SANTOS

VISUALIZAÇÃO DE MEMÓRIA UTILIZANDO JAVA REFLECTION

MOSSORÓ - RN
2013

SAIRO RAONÍ DOS SANTOS

VISUALIZAÇÃO DE MEMÓRIA UTILIZANDO JAVA REFLECTION

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação – associação ampla entre a Universidade do Estado do Rio Grande do Norte e a Universidade Federal Rural do Semi-Árido, para a obtenção do título de Mestre em Ciência da Computação.

Orientadora: Profa. Dra. Angélica Félix de Castro – UFERSA.

MOSSORÓ – RN
2013

SAIRO RAONÍ DOS SANTOS

VISUALIZAÇÃO DE MEMÓRIA UTILIZANDO JAVA REFLECTION

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.

APROVADA EM: _____/_____/____

BANCA EXAMINADORA

_____
Profª. Dra. Angélica Félix de Castro (UFERSA)
Presidente

_____
Prof. Dr. Umberto Souza da Costa (UFRN)
Membro Externo

_____
Prof. Dr. Judson Santos Santiago (UFERSA)
Membro Interno

_____
Prof. Dr. Tiberius de Oliveira e Bonates (UFERSA)
Membro Interno

À minha avó (*in memoriam*).

RESUMO

Neste trabalho, propomos uma abordagem para a geração de uma visualização da memória utilizada por aplicações Java. A abordagem foi implementada em uma ferramenta de visualização de software, que será disponibilizada sem custos pelos autores. Usando a API Java Reflection, inspecionamos a memória da Máquina Virtual Java em tempo de execução. Os dados assim obtidos são interpretados como a descrição de um grafo direcionado no qual vértices são os objetos da aplicação, e as arestas são as referências não-nulas contidas em tais objetos. Uma representação visual do grafo direcionado é então gerada utilizando um software de visualização de grafos. Além da extração e geração de visualizações automáticas, a ferramenta tem funcionalidades que permitem a personalização de tais representações visuais: filtros de destaque, que isolam elementos específicos para melhor visualização; e opções de agrupamento e simplificação que podem ser usadas para reduzir a complexidade visual da imagem resultante. Conduzimos dois casos de estudo com aplicações que utilizam estruturas de dados complexas, através dos quais algumas das funcionalidades propostas são ilustradas.

Palavras-chave: Visualização de software. Visualização de informação. Grafo de memória.

ABSTRACT

In this paper we propose an approach for generating a visualization of the memory used by Java applications. We implemented the approach into a software visualization tool, which is freely available from the authors. Using the Java Reflection API we inspect the memory content of the Java Virtual Machine at runtime. The data thus obtained is regarded as describing a digraph, in which vertices are the application's objects and edges are non-null references contained in those objects. A visual representation of that digraph is then generated using standard graph visualization software. Besides automatic graph extraction and visual representation generation, the tool has novel features that allow for customization of such visual representations: highlighting filters, which isolate specific elements for easy viewing; and grouping and simplifying options that can be used to reduce visual complexity. We conducted two case studies with applications that feature complex data structures, by means of which some of the proposed features are illustrated.

Key words: Information visualization. Software visualization. Memory graph.

LISTA DE FIGURAS

Figura 1 – Modelo de referência para a Visualização 11
Figura 2 – Mapeamento de valores numéricos ao comprimento de barras 20
Figura 3 – Processo de Visualização 21
Figura 4 – Baía de Passamaquoddy 22
Figura 5 – Um mapa dos Estados Unidos mostrando o número de mortes relacionadas ao câncer entre 1970 e 1994 23
Figura 6 – Exemplos de Elementos Gráficos 27
Figura 7 – Exemplos de Propriedades Gráficas 27
Figura 8 – (a) Grafo original (b) Utilização da ferramenta *MagnetViz* 29
Figura 9 – Níveis de Abstração 35
Figura 10 – Um dos diferentes tipos de visualização possíveis na ferramenta *Jinsight*. 38
Figura 11 – *SoftArchViz*, Um Sistema de Visualização da Arquitetura de Softwares 39
Figura 12 – Exemplo de Visualização de Software gerado pela ferramenta *EvoSpaces* 40
Figura 13 – Análise de Memória com a ferramenta DYMEM 41
Figura 14 – Algoritmo utilizado pelo Reflector para observar a memória de uma aplicação . 45
Figura 15 – Árvore binária de busca 46
Figura 16 – Algoritmo utilizado pelo Reflector com a utilização de filtros de seleção 47
Figura 17 – Exemplo dos Elementos Gráficos utilizados nas visualizações geradas pela ferramenta Reflector. 50
Figura 18 – Exemplo da aplicação de filtros 51
Figura 19 – Ilustração de Simplificações Automáticas 52
Figura 20 – Diferença entre grafos de memória 53
Figura 21 – Arquitetura do Reflector 55
Figura 22 – Como utilizar o Reflector 57
Figura 23 – Como utilizar a diferença entre grafos 57
Figura 24 – Tela inicial do Reflector 58
Figura 25 – Janelas de diálogo do Reflector 59
Figura 26 – Outras janelas de diálogo do Reflector 59
Figura 27 – Opções de configuração de filtros 60
Figura 28 – Um grafo de memória é mostrado no Reflector 61
Figura 29 – Árvore digital com filtros de destaque e agrupamento 63
Figura 30 – Trechos de código-fonte da aplicação 63
Figura 31 – Classificador REPTree do WEKA 64

LISTA DE TABELAS

Tabela 1 – Definições de dados, informação e conhecimento no espaço perceptual 15
Tabela 2 – Definições de dados, informação e conhecimento no espaço computacional 15

SUMÁRIO

1 INTRODUÇÃO 11
1.1 PROPOSTA 12
1.2 ORGANIZAÇÃO DO DOCUMENTO 13

2 REFERENCIAL TEÓRICO 14
2.1 DADOS, INFORMAÇÃO E CONHECIMENTO 14
2.2 EVOLUÇÃO DE SOFTWARE 16
2.3 VISUALIZAÇÃO 19
2.3.1 Visualização de Informação 23
2.3.1.1 Pré-processamento e Transformação de Dados 25
2.3.1.2 Mapeamento Visual 25
2.3.1.3 Visões 28
2.3.1.4 Aplicações 28
2.3.2 Visualização de Software 32

3 REFLECTOR 36
3.1 TRABALHOS RELACIONADOS 36
3.2 JAVA REFLECTION 42
3.3 GRAFO DE MEMÓRIA 43
3.3.1 Slicing 44
3.3.2 Extração do Grafo de Memória 44
3.3.3 Filtro de Seleção 46
3.4 PROCESSO DE CRIAÇÃO 47
3.4.1 Pré-processamento e Transformação de Dados 48
3.4.2 Mapeamento Visual 48
3.4.3 Visões 50
3.5 VISUALIZAÇÃO NO REFLECTOR 50
3.5.1 Filtro de Destaque 51
3.5.2 Simplificações Automáticas 52
3.5.3 Diferença entre Grafos de Memória 53
3.6 ARQUITETURA 54
3.7 UTILIZAÇÃO DO REFLECTOR 55

3.7.1 Posicionamento no Ciclo de Desenvolvimento 55
3.7.2 Utilização da Interface Gráfica 56
3.8 SÍNTESE 61

4 ESTUDO DE CASO 62
4.1 TECLADO VIRTUAL 62
4.2 CLASSIFICADOR WEKA 64
4.3 RESULTADOS 65

5 CONCLUSÃO 66
5.1 CONTRIBUIÇÕES 66
5.2 TRABALHOS FUTUROS 67

REFERÊNCIAS 68

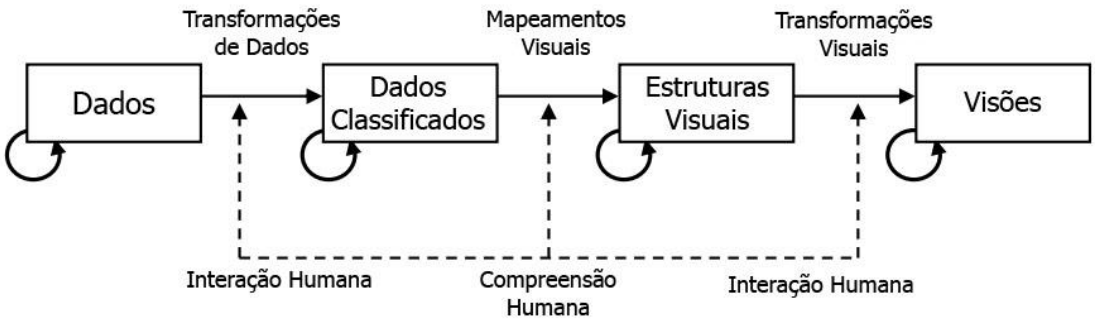
1 INTRODUÇÃO

Grandes sistemas de software sofrem mudanças frequentes durante seu ciclo de vida. Tais mudanças são executadas como tarefas de manutenção e evolução, que procuram corrigir erros ou adaptar o sistema a novos requisitos. Tentativas de estimar os custos da manutenção e evolução de um software normalmente informam que essas fases do ciclo de vida concentram de 50 a 75% dos custos totais de um software (PINZGER, 2005).

Grande parte do esforço feito por desenvolvedores nessas tarefas diz respeito à compreensão do software que está sendo mantido, seja ela em relação a sua arquitetura e aspectos de projeto ou sua implementação propriamente dita. Desenvolvedores precisam construir modelos mentais que mostrem esses aspectos. A visualização de características abstratas de informações de baixo nível tem sido bem aceita como uma forma útil de auxiliar desenvolvedores na construção de tais modelos mentais (PINZGER, 2005).

A visualização é o mapeamento de dados para uma forma visual que pode ser compreendida pelo homem. A Figura 1, adaptada de Shneiderman (1999), descreve esse mapeamento e serve como um simples modelo de referência para a visualização. Nessa figura, o fluxo de dados passa por uma série de transformações. O usuário de uma ferramenta de visualização pode ajustar essas transformações, usando controles de interface, para abordar uma determinada tarefa (MALETIC, 2003).

Figura 1 – Modelo de referência para a Visualização.



Fonte: SHNEIDERMAN (1999).

A visualização de software se encaixa diretamente nesse modelo. Os dados de entrada podem ser código-fonte, informações sobre a execução, documentos de projeto, etc (MALETIC, 2003). Gallagher *et al.* (2008) definem a visualização de software como o processo de mapear entidades no domínio de um sistema de software para representações

gráficas, buscando auxiliar a compreensão e o desenvolvimento do mesmo. Assim, o principal objetivo da área é criar imagens de softwares usando objetos visuais que podem representar, por exemplo, sistemas e seu comportamento em tempo de execução. Como resultado, desenvolvedores podem ter uma melhor percepção de como o software está estruturado, compreender seu funcionamento e explicar e comunicar seu desenvolvimento. Representações gráficas eficazes podem fornecer uma forma de apresentação de informação que é mais próxima do modelo mental dos usuários que representações textuais, e tiram vantagem de suas capacidades perceptivas.

Assim, de acordo com Gallagher *et al.* (2008), a visualização de software pode proporcionar uma redução no custo de desenvolvimento de um software e de sua evolução. Pode ainda apoiar a evolução do software ao auxiliar tomadores de decisão na compreensão do mesmo em diferentes níveis de abstração e diferentes pontos do ciclo de vida de um sistema.

1.1 PROPOSTA

Neste trabalho é descrita uma abordagem que se propõe a analisar a memória usada por uma aplicação Java e representá-la visualmente. Isto é, nesse caso, a visualização utiliza dados obtidos em tempo de execução para gerar uma representação visual da memória utilizada por uma aplicação Java em um momento de sua execução. Para isso, é utilizada a API Java Reflection, que possibilita o acesso à memória utilizada por uma aplicação em tempo de execução, permitindo: extrair informação sobre o uso da memória de uma aplicação em qualquer momento de sua execução; determinar a classe de um objeto; e acessar os atributos de um objeto, assim como obter seus valores. O trabalho inclui ainda a implementação da abordagem em uma ferramenta homônima. A ferramenta fornece ao usuário uma representação visual em forma de grafo de todas as estruturas de dados e variáveis que podem ser acessadas a partir de um determinado objeto, além das relações entre essas estruturas e variáveis, em um determinado momento da execução da aplicação. Assim, é possível visualizar toda a memória usada pelo programa que está, em um determinado momento, acessível a partir de um objeto em particular.

Tal funcionalidade permite que um desenvolvedor detecte visualmente várias características da aplicação que podem não ser facilmente perceptíveis quando apenas seu

código-fonte é analisado. Ela pode ser útil também para tarefas de depuração e documentação, ou mesmo simplesmente ajudar desenvolvedores a compreender os relacionamentos existentes entre objetos da aplicação.

1.2 ORGANIZAÇÃO DO DOCUMENTO

O trabalho está estruturado da seguinte forma:

- No segundo capítulo é feita uma introdução aos conceitos básicos de dados, informação e conhecimento e uma discussão breve sobre a área da visualização e suas subáreas relevantes para o presente trabalho.
- O terceiro capítulo apresenta uma descrição detalhada sobre o funcionamento da ferramenta desenvolvida, explicando suas diversas funcionalidades e sua forma de utilização.
- O quarto capítulo aborda dois estudos de caso feitos para ilustrar a utilização da ferramenta em aplicações reais.
- As conclusões do trabalho encontram-se no quinto capítulo.

2 REFERENCIAL TEÓRICO

Hoje em dia, o mundo observa uma quantidade crescente de dados que devem ser assimilados diariamente. A globalização da economia e da comunicação e, acima de tudo, os avanços tecnológicos, têm trazido o que a literatura define como poluição de informação (CAMERON, 2000). O desenvolvimento constante de dispositivos de armazenamento e formas de criar e coletar dados tem influenciado a forma como eles são manipulados e, na maioria dos casos, tais dados são armazenados sem nenhum tipo de filtragem ou refinamento para utilização posterior. Desta forma, o que acontece na realidade não é uma explosão de informação, mas sim uma explosão de dados. Esses dados precisam ser analisados para que possam gerar informação (KEIM *et al.*, 2010; MAZZA, 2009).

Com essa análise, o fluxo contínuo e constante de dados ao qual somos ativa ou passivamente submetidos diariamente pode fornecer uma grande quantidade de informação (MAZZA, 2009). Assim, o problema que enfrentamos não é a aquisição de dados, mas sim a habilidade de identificar métodos e modelos que sejam capazes de transformar tais dados em conhecimento compreensível e confiável (KEIM *et al*., 2010).

2.1 DADOS, INFORMAÇÃO E CONHECIMENTO

Como é possível ler dados, compreender informações e adquirir conhecimento, é necessário diferenciar esses termos no espaço perceptual e cognitivo. Porém, com o advento da tecnologia, tornou-se possível armazenar também dados, informação e conhecimento em forma digital, trazendo definições distintas para tais termos também no espaço computacional (CHEN *et al.*, 2009). A Tabela 1 traz as definições de Ackoff (1989) para o espaço perceptual e cognitivo.

Considere que P é o conjunto de toda a memória humana explícita e implícita. A memória explícita guarda lembranças de eventos, fatos e conceitos e a compreensão de seus significados, contextos e associações, enquanto a memória implícita contém todas as formas não-conscientes de memória, como respostas emocionais, habilidades e hábitos. É possível, assim, separar a memória em três subconjuntos, $P^{dados} \subset P$, $P^{info} \subset P$ e $P^{con} \subset P$, onde

P^{dados}, P^{info} e P^{con} são os conjuntos de toda a memória explícita e implícita possível sobre dados, informações e conhecimento, respectivamente (CHEN *et al.*, 2009).

Tabela 1 – Definições de dados, informação e conhecimento no espaço perceptual e cognitivo.

Categoria	**Definição**
Dados	Dados são símbolos. Simplesmente existem e não têm significado algum além de sua existência. Podem existir em qualquer forma, utilizável ou não.
Informação	Informações são dados processados, fornecendo respostas para as perguntas "quem", "o quê", "onde" e "quando". São basicamente dados aos quais foram atribuídos significados por meio de alguma conexão relacional. Esses significados podem ser úteis ou não.
Conhecimento	Conhecimento é a aplicação de dados e informações, fornecendo respostas à pergunta "como". É uma coleção coesa e coerente de informações de forma que seu objetivo é ser útil em algum contexto ou tarefa.

Fonte: ACKOFF (1989).

Considere também que C é o conjunto de todas as representações possíveis em memória computacional. Similarmente, é possível considerar três subconjuntos de representações, C^{dados}, C^{info} e C^{con}, onde $C^{dados} \subset C$, $C^{info} \subset C$ e $C^{con} \subset C$. A Tabela 2 mostra as definições de dados, informação e conhecimento no espaço computacional de acordo com (CHEN *et al.*, 2009).

Tabela 2 – Definições de dados, informação e conhecimento no espaço computacional.

Categoria	**Definição**
Dados	Representações computadorizadas de modelos e atributos de entidades reais ou simuladas.
Informação	Dados que representam os resultados de processos computacionais como análises estatísticas ou transcrições de significados atribuídos por seres humanos.
Conhecimento	Dados que representam os resultados de um processo cognitivo simulado pelo computador, como percepção, aprendizagem, associação e raciocínio, ou transcrições de conhecimento adquirido por seres humanos.

Fonte: CHEN *et al.* (2009).

2.2 EVOLUÇÃO DE SOFTWARE

Um processo de software é um conjunto de atividades e resultados associados que geram um produto de software. Essas atividades são, em sua maioria, executadas por engenheiros de software. Há quatro atividades de processo comuns a todos os processos de software. Essas atividades são:

- Especificação de software: a funcionalidade do software e as restrições em sua operação devem ser definidas.
- Desenvolvimento de software: o software deve ser produzido de modo que atenda a suas especificações.
- Validação do software: o software deve ser validado para garantir que ele faz o que o cliente deseja.
- Evolução do software: o software deve evoluir para atender às necessidades mutáveis do cliente.

Diferentes processos de software organizam essas atividades de maneiras diversas e são descritos em diferentes níveis de detalhes. Os prazos das atividades variam, do mesmo modo que os resultados de cada atividade. Diferentes organizações podem utilizar processos diferentes para produzir o mesmo tipo de produto (SOMMERVILLE, 2011).

Um modelo de processo de software é uma descrição simplificada de um processo de software, que é apresentada de uma perspectiva específica. Os modelos, por natureza, são simplificações e, assim, um modelo de processo de software é uma abstração do processo real que está sendo descrito. Existe uma série de diferentes modelos gerais, ou paradigmas, de desenvolvimento de software:

- Cascata: considera as atividades da criação de um software e as representa como fases separadas do processo, como especificações de requisitos, projeto de software, implementação, testes, etc. Após cada estágio ter sido definido, ele é ‘aprovado’ e o desenvolvimento prossegue para o estágio seguinte.
- Desenvolvimento evolucionário: intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é rapidamente desenvolvimento a partir de especificações abstratas. Em seguida, ele é refinado com informações do cliente, a fim de produzir um sistema que satisfaça suas necessidades.

- Transformação formal: se baseia na produção de uma especificação formal matemática do sistema e na transformação dessa especificação, utilizando-se de métodos matemáticos em um programa.
- Montagem a partir de componentes reutilizáveis: supõe que partes do sistema já existem. O processo de desenvolvimento se concentra na integração dessas partes, em vez de desenvolvê-las a partir do zero.

Todos os diferentes modelos concordam, porém, em um ponto que é comum a todos eles: os grandes custos da fase de evolução do software. Além dos custos de desenvolvimento, também existem os custos de alteração do software, o que é feito na fase de evolução. Para muitos sistemas de software com um longo ciclo de duração, é provável que esses custos excedam o custo de desenvolvimento em três ou quatro vezes (SOMMERVILLE, 2011).

Parnas (1994) declarou sobre o envelhecimento de software:

> "Programas, como pessoas, envelhecem. Não é possível evitar o envelhecimento, mas podemos compreender suas causas, tomar providências para limitar seus efeitos, reverter temporariamente os danos causados, e nos preparar para o dia quando o software não for mais viável."

O fenômeno do envelhecimento de um software se refere ao acúmulo de erros durante o ciclo de vida do software com o tempo, resultando em falhas. Uma gradual degradação em seu desempenho pode acompanhar o envelhecimento (GARG *et al.*, 1998).

Há dois tipos de envelhecimento de software. O primeiro é causado pela negligência dos desenvolvedores em relação à modificação do produto para atender a novas necessidades dos usuários; o segundo é resultado das mudanças que são feitas para atender tais necessidades. Os sintomas do fenômeno são semelhantes àqueles do envelhecimento humano: softwares em processo de envelhecimento são cada vez mais difíceis de atualizar para acompanhar exigências de mercado; muitas vezes sofrem degradação em seu desempenho, resultado de uma estrutura gradualmente deteriorada; e passam a ter *bugs*, pois erros são introduzidos quando mudanças são feitas (PARNAS, 1994).

A evolução de software é a área de pesquisa que aborda a forma como um software pode mudar para reagir às mudanças do ambiente onde é utilizado para que ele não se torne obsoleto. Ela inclui todos os estágios da vida de um software: desenvolvimento, manutenção, migração e retirada. A evolução se refere não apenas às mudanças que se tornam necessárias

para evitar que um software se torne ultrapassado, mas também às causas e efeitos de tais mudanças (PINZGER, 2005).

Tradicionalmente, a evolução de software acontecia após o desenvolvimento e implantação de um sistema, ao ponto de até ter orçamentos, equipes e procedimentos diferentes para tal. Porém, com a mudança e crescente competitividade no mercado de desenvolvimento de software, a natureza da evolução de software tornou-se um processo contínuo, no qual não há divisão entre desenvolvimento e evolução (BOEHM; BECK, 2010).

Lehman *et al.* (1997) enumeram um conjunto de leis que chamam de 'Leis da Evolução de Software' e apresentam algumas observações gerais:

- Mudanças Contínuas: sistemas devem ser continuamente adaptados, ou se tornam progressivamente menos satisfatórios;
- Complexidade Crescente: com a evolução, a complexidade do sistema cresce, a não ser que sejam tomadas providências para mantê-la ou diminuí-la;
- Conservação de Familiaridade: desenvolvedores e usuários devem manter sua familiaridade com o conteúdo e comportamento do sistema para que uma evolução satisfatória seja atingida;
- Crescimento Contínuo: o conteúdo funcional de um sistema deve crescer continuamente para manter a satisfação do usuário durante seu tempo de vida.
- Declínio de Qualidade: a qualidade do sistema parecerá estar em declínio, a não ser que ele seja rigorosamente mantido e adaptado a mudanças operacionais do ambiente.

Um dos maiores problemas com a evolução de sistemas de software é que, quando eles são adaptados a mudanças em seus requisitos, sua arquitetura, projeto e implementação podem ser atingidas. Os sinais de envelhecimento causados por tais impactos são, por exemplo, a erosão ou incompatibilidade de arquitetura, que são definidos por Perry e Wolf (1992) como "violações na arquitetura que levam a crescentes problemas e fragilidade em sistemas".

As principais causas dos sinais de envelhecimento são exatamente as más decisões de projeto, e mudanças que prejudicam a arquitetura ou a falta de conformidade entre a implementação e a arquitetura pretendida. Os efeitos de tais sinais são uma queda na produtividade e na qualidade do sistema. Consequentemente, os custos de conserto de tais problemas e seu impacto no processo de adaptação do sistema a novos requisitos explodem, e a vida do software é encurtada (PINZGER, 2005).

Para superar ou evitar os efeitos negativos do envelhecimento e da evolução, é necessário mudar o processo de desenvolvimento e manutenção. É necessário fornecer melhor apoio à mudança e evolução do software, incluindo o apoio e suporte à análise de sistemas de software e sua evolução e ao controle e execução de mudanças (PINZGER, 2005). Nesse contexto, o foco deste trabalho é a análise do software como suporte para o controle da evolução. O suporte proposto baseia-se em técnicas de visualização da estrutura do software sendo analisado.

2.3 VISUALIZAÇÃO

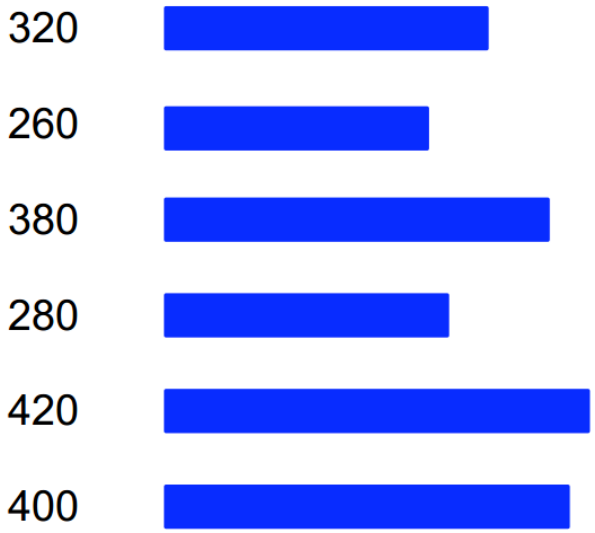
Conforme apontado por Hutchins e Lintern (1995), o raciocínio não é algo que acontece apenas dentro da cabeça das pessoas; pouco trabalho intelectual é feito com os olhos e ouvidos fechados. Maior parte da cognição e compreensão acontece com algum tipo de interação com ferramentas cognitivas, como lápis e papel, calculadoras e, cada vez mais, computadores. Desde o início da ciência, diagramas, notações matemáticas e a escrita têm sido ferramentas essenciais para o cientista. Agora já existem ferramentas analíticas poderosas e interativas, como o MATLAB, Maple, Mathematica, além de diversos tipos de bancos de dados. Áreas de pesquisa inteiras, como a genômica e a proteômica, baseiam-se em ferramentas computacionais de análise e armazenamento (WARE, 2012).

Isso acontece porque, como o homem é capaz de perceber representações imagéticas muito bem, é possível representar grandes quantidades de dados ao mapeá-los a diferentes atributos visuais. Características visuais como cor, formato, tamanho e movimento são imediatamente absorvidas e processadas pela habilidade perceptual da visão, antes mesmo que os processos cognitivos da mente humana possam ser considerados (MAZZA, 2009). Visualizações têm um papel pequeno, porém crucial e crescente, em sistemas cognitivos. Representações gráficas fornecem o melhor canal possível de transferência de informação entre um objeto e um ser humano, pois se adquire mais informação através da visão do que através de todos os outros sentidos combinados. Os 20 bilhões de neurônios utilizados pelo cérebro para analisar informações visuais fornecem um mecanismo de busca de padrões que é um componente fundamental em maior parte das atividades cognitivas (WARE, 2012).

A Figura 2, retirada de Mazza (2009), traz um exemplo que ajuda a esclarecer essa ideia. Considere que deve-se determinar o mínimo e o máximo valor numérico indicado à

esquerda. Se as barras não existissem, o seguinte procedimento seria necessário para resolver esse problema: ler cada um dos valores; memorizar os valores extremos que aparecem ao lê-los até o final. Se o exercício é repetido com a ajuda das barras à esquerda, o tamanho delas mostra instantaneamente quais são os valores extremos. Essa informação é processada pela percepção visual, que reconhece imediatamente os comprimentos das barras e os organiza em relação aos valores representados. Seria possível também representar as barras com diferentes cores ou diferentes larguras para codificar outros dados. Nesse caso, as representações visuais, se bem construídas, podem ser úteis não apenas para uma percepção mais rápida de informações, mas também para o processamento de vários itens de informação ao mesmo tempo (MAZZA, 2009).

Figura 2 – Mapeamento de valores numéricos ao comprimento de barras.



Fonte: MAZZA (2009).

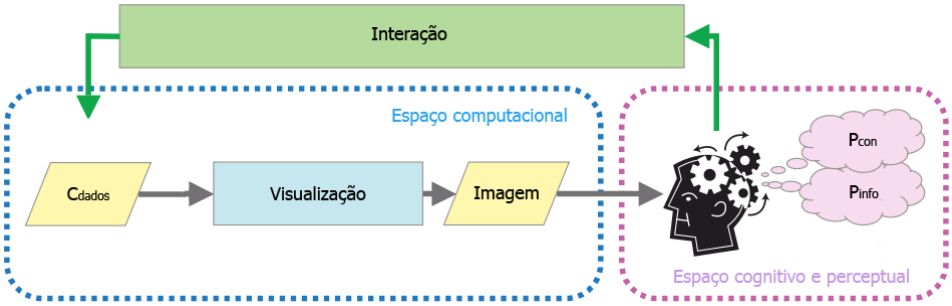
De acordo com Ware (2012), o termo visualização significava, até recentemente, construir uma imagem visual na mente. O termo tem passado a significar, porém, algo mais próximo de uma representação gráfica de dados ou conceitos. Isto é, a visualização se tornou um artefato externo que apóia a tomada de decisões, e não mais uma construção interna da mente humana.

Outras diferentes definições podem ser encontradas na literatura. Por exemplo, de acordo com McCormick *et al.* (1987), a visualização é um método de computação que transforma aquilo que é simbólico em algo geométrico, possibilita que pesquisadores observem suas simulações e computações, oferece um método para ver o que está oculto,

enriquece o processo da descoberta científica, e cria compreensões profundas e inesperadas. Já Zhang (2008) define a visualização como o processo de transformar dados, informação e conhecimento em representações gráficas para apoiar tarefas como a análise de dados, a exploração de informações, a explanação de informações, a previsão de tendências, a detecção de padrões, a descoberta de ritmos, entre outras.

A Figura 3, adaptada de Chen *et al.* (2009), mostra um processo de visualização comum, ilustrando instâncias de dados, informação e conhecimento no espaço computacional e no espaço perceptual e cognitivo. Dado um conjunto de dados C^{dados}, o usuário toma decisões sobre quais ferramentas de visualização deseja usar para explorar o conjunto de dados. Ele experimenta então com diferentes controles, como estilos, leiautes, posições e cores até obter um resultado satisfatório.

Figura 3 – Processo de Visualização.



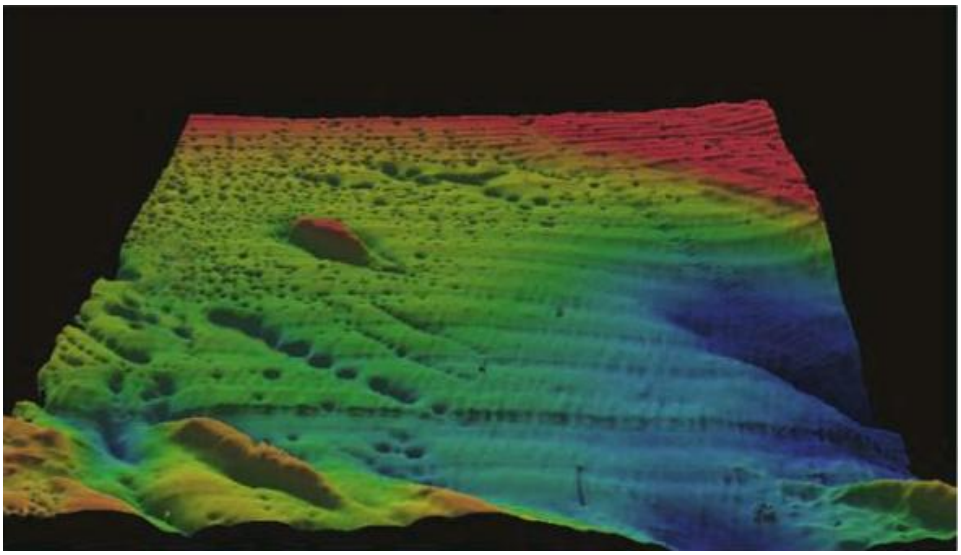
Fonte: CHEN *et al.* (2009).

Um dos maiores benefícios da visualização é a quantidade de informação que pode ser rapidamente interpretada se ela é apresentada bem. A Figura 4, retirada de Ware (2012), mostra uma visualização derivada de um estudo exploratório feito na Baía de Passamaquoddy, nos Estados Unidos, onde as marés são as mais altas do mundo. Aproximadamente um milhão de medições foram feitas. Tradicionalmente, esse tipo de dado é apresentado em forma de tabela náutica. Porém, quando os dados são convertidos para um mapa de altura e mostrados utilizando técnicas de computação gráfica, várias características que não podem ser percebidas facilmente na tabela se tornam visíveis. Um padrão de pequenos sulcos redondos chamados *pockmarks* pode ser visto imediatamente, e é fácil perceber que eles formam linhas (WARE, 2012).

A Figura da Baía de Passamaquoddy destaca uma série de vantagens da visualização:

- A visualização fornece a habilidade de compreender quantidades enormes de dados. A informação importante a se retirar de um milhão de medições se tornou disponível imediatamente;
- A visualização permite a percepção de propriedades emergentes que não foram previstas. Nessa visualização, o fato de que as *pockmarks* aparecem em linhas é imediatamente evidente. A percepção de um padrão como esse pode muitas vezes ser a base para novas descobertas;
- A visualização muitas vezes permite que problemas com os dados se tornem aparentes imediatamente. A visualização normalmente revela não apenas informações sobre os dados, mas também sobre a forma como os dados foram coletados. Com uma visualização adequada, erros dos dados se tornam visíveis imediatamente. Por esse motivo, visualizações têm valor incalculável para processos de controle de qualidade;
- A visualização facilita a compreensão de características de qualquer escala nos dados.

Figura 4 – Baía de Passamaquoddy.



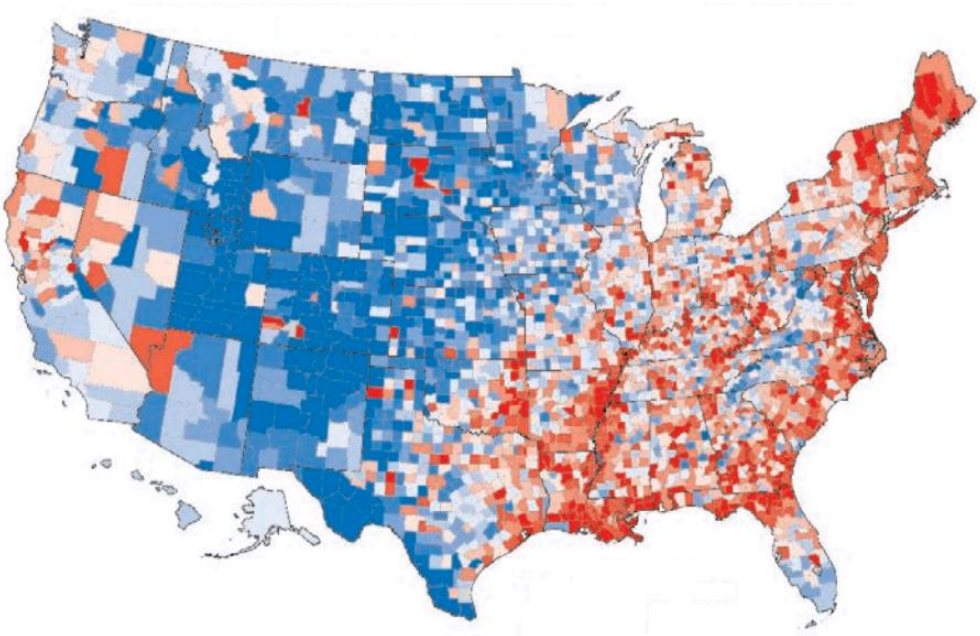
Fonte: WARE (2012).

Tais vantagens tornam a visualização uma abordagem interessante para tarefas de análise exploratória de dados, uma das aplicações que mais se beneficiam do uso de representações visuais e da habilidade do ser humano de analisar informação através da

percepção visual. A visualização têm sido utilizada há muitos anos por tais aplicações para identificar propriedades, relacionamentos, regularidades ou padrões (MAZZA, 2009).

Um exemplo da utilização da visualização para análise exploratória de dados pode ser visto na Figura 5, retirada de Mazza (2009). Ela mostra estatísticas de mortalidade relacionada ao câncer entre 1970 e 1994 nos Estados Unidos. Nela, distritos são representados por cores que vão do azul ao vermelho, de acordo com a porcentagem de casos encontrados em cada distrito. Devido às cores, é possível identificar a incidência média de casos nas áreas: distritos coloridos de branco têm incidência média, distritos de incidência baixa são mostrados em azul e aqueles com incidência alta são representados em vermelho. Assim, torna-se imediatamente óbvio para o observador quais são as partes do país onde se concentra maior parte das regiões com alta mortalidade. A imagem não fornece uma explicação sobre por que a incidência de mortes é maior em certas partes do país que em outras, mas pode sugerir que pesquisadores conduzam experimentos em determinadas regiões, o que pode revelar fatores que aumentam o risco de desenvolvimento de câncer (MAZZA, 2009).

Figura 5 – Um mapa dos Estados Unidos mostrando o número de mortes relacionadas ao câncer entre 1970 e 1994.



Fonte: MAZZA (2009).

2.3.1 Visualização da Informação

Em linhas gerais, a visualização pode ser dividida em duas categorias: a visualização científica e a visualização de informação. A visualização científica é muitas vezes usada como um aumento do sistema sensorial humano ao mostrar coisas que acontecem em velocidades muito altas ou muito baixas para a percepção do olho, ou estruturas muito menores ou maiores que a escala humana, ou fenômenos como o raio X e a radiação infravermelha, que não podem ser sentidos diretamente (ZHANG, 2008).

Já a visualização de informação desenvolve métodos para visualizar dados abstratos onde referências espaciais explícitas não são dadas. Exemplos comuns incluem dados sobre transações financeiras, redes sociais, entre outros. Dados de tais áreas não apenas são grandes, mas muitas vezes têm também centenas de dimensões. Além disso, em adição a tipos de dados numéricos e textuais, algumas dessas dimensões podem ser de tipos de dados complexos como imagens, vídeos e áudios, então os valores dos dados não podem ser mapeados naturalmente em um espaço 2D ou 3D, como acontece na visualização científica (KEIM *et al.*, 2010).

A visualização de informação é definida por Card, Mackinglay e Shneiderman (1999) como "o uso de representações visuais interativas e apoiadas por computador de dados abstratos para ampliar a cognição."

De acordo com Moere (2008), um método de visualização de informação caracteriza-se unicamente por um conjunto predefinido de regras de mapeamento de dados. Cada regra determina como um certo elemento de dados, contendo atributos, valores de dados e relacionamentos com outros elementos de dados, é transportado para uma forma gráfica. Geralmente, cada elemento de dados corresponde a um único elemento visual, como um ponto, linha ou forma, que é então alterado por operações de transformação de acordo com o valor de dados que contém. A natureza exata dessas regras tende a ser determinada pelo desenvolvedor da aplicação, que procura otimizar a eficácia, a exatidão e completude do gráfico final para o propósito final do usuário, e a eficiência, isto é, a relação entre os recursos utilizados e os critérios de eficácia (MOERE, 2008).

Softwares dedicados à criação de representações visuais de dados abstratos, mesmo que difiram bastante entre si, seguem um mesmo processo de geração. Como mencionado anteriormente, são utilizados dados abstratos, e esse tipo de dado não tem uma conexão específica com o espaço físico. Por exemplo, eles podem tratar de nomes próprios, preços de produtos, resultados de votações, etc. Tais dados raramente são encontrados em um formato apropriado para tratamento com ferramentas automáticas de processamento e, em particular,

softwares de visualização. Portanto, eles precisam ser processados apropriadamente, antes de serem representados graficamente (MAZZA, 2009).

De acordo com Mazza (2009), a criação de um artefato visual é um processo que pode ser modelado através de uma sequência de estágios sucessivos: o preprocessamento e transformação de dados; o mapeamento visual; e a criação de visões.

2.3.1.1 Preprocessamento e Transformação de Dados

O termo "dados crus" é utilizado para descrever dados fornecidos pelo mundo ao nosso redor. Esses dados podem ser gerados por ferramentas de medição, ou calculados por softwares apropriados, como acontece com dados de previsão meteorológica. Outro exemplo interessante é o dos dados relacionados a eventos e entidades mensuráveis, como o número de habitantes ou a taxa de natalidade das cidades de um estado. Em cada um desses casos, essas coleções de dados raramente são fornecidas com uma estrutura lógica precisa. A estrutura comumente utilizada para esse tipo de dado é tabular, em um formato apropriado para o software que deve recebê-lo e processá-lo. Em outras ocasiões, os dados de entrada estão em um ou mais banco de dados e estão, portanto, disponíveis em formato eletrônico e estruturados de forma bem definida. Nesse caso, os dados crus correspondem aos dados localizados nos bancos de dados, e a fase de preprocessamento envolve a extração dos dados de tais bancos de dados e a conversão deles para o formato usado pelo software de visualização.

O preprocessamento e transformação dos dados consistem basicamente nessa conversão de dados crus para um formato que seja apropriado para a utilização pelo software de visualização. As operações mais comuns nessa fase são: a filtragem, que é feita para eliminar dados desnecessários; o cálculo utilizando dados crus para a obtenção de novos dados, como estatísticas que possam ser representadas na versão visual dos dados; e a adição de atributos (ou metadados) aos dados, que podem ser utilizados para organizá-los logicamente.

2.3.1.2 Mapeamento Visual

Os principais problemas do processo de criação de visualizações se encontram na definição de quais estruturas visuais usar para mapear os dados e sua localização na área de exibição. Como mencionado anteriormente, dados abstratos não têm necessariamente uma localização real no espaço físico. Há alguns tipos de dados abstratos que, por natureza, podem ser atrelados facilmente a uma localização espacial. É o caso de dados obtidos por estações de monitoramento de poluição atmosférica. É fácil encontrar uma posição em um mapa para tais dados, já que as estações de monitoramento que fazem as medições estão situadas em um lugar específico no espaço. É o mesmo caso de dados retirados de entidades que têm uma estrutura topológica definida, como aqueles relacionados ao tráfego de dados de uma rede de computadores. Porém, há diversos tipos de dados que pertencem a entidades que não têm nenhum posicionamento geográfico ou topológico natural. Por exemplo, as referências bibliográficas de artigos científicos, o consumo de gasolina de um carro, ou o salário de vários funcionários dentro de uma companhia. Esse tipo de dado não tem uma correspondência imediata com as dimensões do espaço físico ao seu redor.

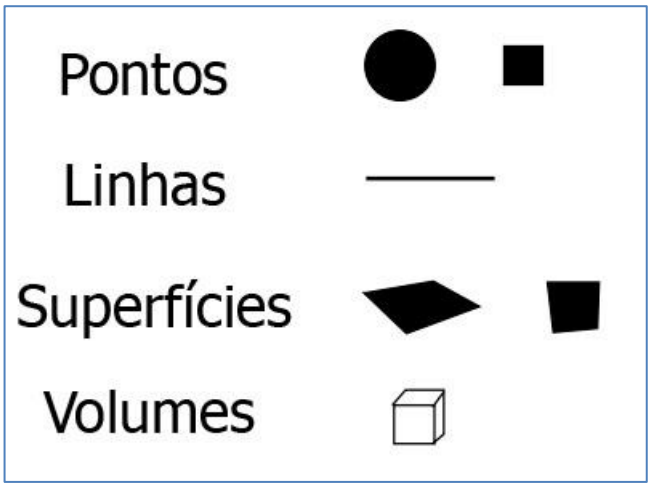
É necessário, portanto, definir as estruturas visuais que corresponderão aos dados que se deseja representar visualmente. Esse processo é chamado de mapeamento visual. O processo consiste na definição de três estruturas: o substrato espacial, os elementos gráficos, e as propriedades gráficas.

O substrato espacial define as dimensões físicas onde a representação visual é criada. O substrato espacial pode ser definido em termos de eixos. Em espaço cartesiano, o substrato espacial corresponde aos eixos x e y. Cada eixo pode ser de tipo diferente, dependendo do tipo de dados que se deseja mapear nele. Em particular, um eixo pode ser quantitativo, onde há uma métrica associada aos valores reportados no eixo; ordinal, quando os valores são espalhados no eixo em uma ordem que corresponde à ordem dos dados; e nominal, onde a região de um eixo é dividida em uma coleção de subregiões sem uma ordem intrínseca.

Os elementos gráficos são todos os objetos visíveis que aparecem em uma representação gráfica. Há quatro tipos de elementos visuais: pontos, linhas, superfícies e volumes. A Figura 6, adaptada de Mazza (2009), mostra exemplos dos quatro tipos.

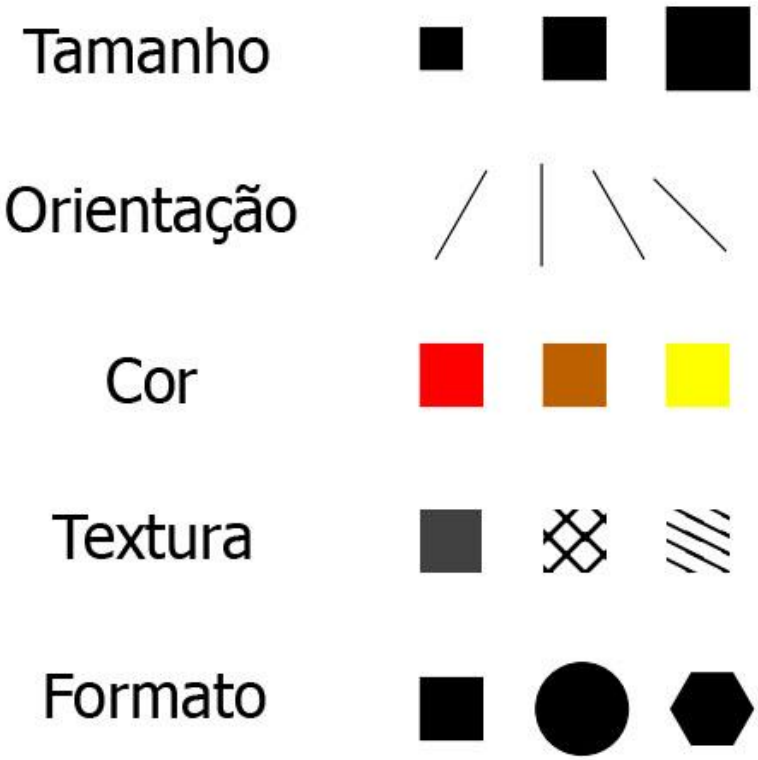
As propriedades gráficas são propriedades dos elementos gráficos às quais a retina do olho humano é muito sensível. Elas são independentes da posição ocupada por um elemento visual no substrato espacial. As propriedades gráficas mais comuns são o tamanho, a orientação, a cor, a textura e o formato. A Figura 7, adaptada de Mazza (2009), mostra exemplos. Tais propriedades são aplicadas aos elementos gráficos e determinam as propriedades da disposição visual que será apresentada na imagem final.

Figura 6 – Exemplos de Elementos Gráficos.



Fonte: MAZZA (2009).

Figura 7 – Exemplos de Propriedades Gráficas.



Fonte: MAZZA (2009).

Em termos da percepção visual do ser humano, todas as propriedades gráficas se comportam da mesma forma. Algumas propriedades gráficas são mais eficazes que outras do ponto de vista de valores quantitativos. A cor pede atenção especial, pois é a única

propriedade gráfica na qual a percepção pode depender de fatores culturais, linguísticos e fisiológicos. Algumas populações, por exemplo, usam um número limitado de termos para definir todo o espectro de cores. É possível, portanto, que pessoas de diferentes culturas utilizem terminologias diferentes para identificar a mesma cor, ou até mesmo tenham percepções diferentes, por não terem um termo específico para definir uma determinada cor. As únicas cores que têm o mesmo nome em todo o mundo são: branco, preto, vermelho, verde, amarelo, e azul.

É necessário também lembrar que grande parte da população tem um problema de percepção particular, o daltonismo. Pessoas que sofrem de tal condição são incapazes de perceber a diferença entre as cores verde e vermelho, ou entre as cores amarelo e azul. É, portanto, necessário considerar que há algumas pessoas com esse problema visual e desenvolver aplicações nas quais é possível mudar o mapeamento de cores.

2.3.1.3 Visões

As visões são o resultado final do processo de geração. Elas são o resultado do mapeamento de estruturas de dados às estruturas visuais, gerando uma representação visual no espaço físico representado pelo computador. São basicamente o que pode ser mostrado na tela do computador.

As visões são caracterizadas por um problema inerente: uma quantidade de dados a representar é muito grande para o espaço disponível. Esse é um problema encontrado frequentemente, já que situações reais muitas vezes envolvem uma quantidade muito grande de dados. Nesses casos, quando a área de exibição é muito pequena para suportar visualmente todos os elementos de uma representação visual, certas técnicas são utilizadas, como o *zoom*, o deslocamento, a rolagem, entre outras.

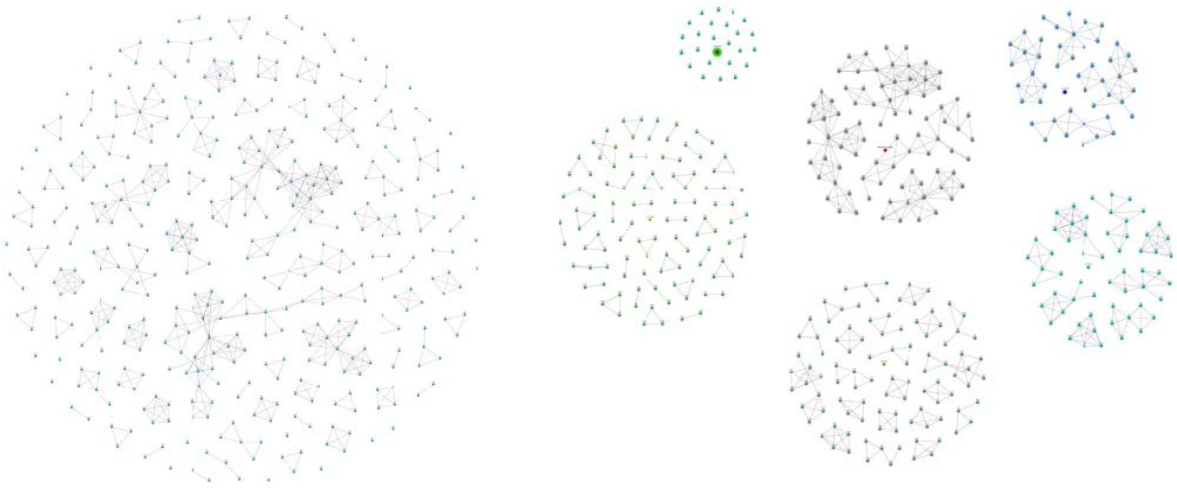
2.3.1.4 Aplicações

A visualização de informação tem aplicações nos mais diversos domínios. Por exemplo, no trabalho de Pinto *et al.* (2012) é feita uma comparação entre ferramentas de

visualização de informação desenvolvidas com o intuito de apoiar instituições de educação superior. A comparação é feita com base em métricas propostas em outros trabalhos da área, como uma taxonomia de ferramentas que apoiam o uso fluente de visualizações proposta por Heer e Shneiderman (2012), e uma proposta de comparação visual para ferramentas de visualização de informação feita por Gleicher *et al.* (2011).

Spritzer e Dal Sasso Freitas (2012) delineam o processo de projeto e avaliação da ferramenta *MagnetViz*, utilizada para gerar representações visuais de grafos. A ferramenta é projetada para a manipulação interativa de grafos, permitindo que o usuário obtenha visualizações com base na topologia do grafo e/ou nos atributos de seus vértices e arestas. Ela permite ainda que o usuário utilize componentes visuais para agrupar vértices e arestas que atendam critérios criados pelo próprio usuário, forçando o grafo a se reorganizar de forma semanticamente mais relevante para um propósito específico. A Figura 8 ilustra a utilização da ferramenta. A Figura 8 (a) mostra um grafo com um *layout* padrão, enquanto a Figura 8 (b) mostra o mesmo grafo com a utilização de uma funcionalidade de agrupamento de vértices de acordo com o tamanho dos componentes conectados existentes no grafo.

Figura 8 – (a) Grafo original (b) Utilização da ferramenta *MagnetViz*.



(a) (b)

Fonte: SPRITZER e DAL SASSO FREITAS (2012).

No trabalho de Ning *et al.* (2012), os autores falam sobre uma aplicação da visualização de informação para a área de Telemedicina. Eles argumentam que, devido à particular complexidade dos dados utilizados na área, a aplicação direta de técnicas de visualização existentes é ineficaz e ineficiente ou até mesmo impossível. O trabalho apresenta

um *framework* integrado para a criação de aplicações de visualização de informação na área da *Telemedicina*.

No trabalho de Tobiasz *et al.* (2009) é descrita a ferramenta Lark, que facilita a coordenação de interações com representações criadas por ferramentas de visualização de informação em espaços digitais compartilhados. Observando as possibilidades de interações de múltiplos usuários com dados simultaneamente ou independendemente, os autores fornecem na ferramenta um ambiente coerente de visualização e interação colaborativa ao fornecer apoio visual e algorítmico à coordenação de atividades de análise de dados em grandes *displays multi-touch* compartilhados.

No trabalho de Rabelo *et al.* (2008), os autores discorrem sobre o uso de técnicas de visualização de informação na representação de dados gerados com o uso de métodos de mineração de dados. Os autores discutem que a área da mineração de dados tem contribuído na busca por conhecimento implícito que pode apoiar a tomada de decisões em diversos domínios, mas que a análise dos resultados obtidos utilizando suas técnicas pode ser difícil. É feita então uma análise de algumas técnicas de visualização de informação, buscando encontrar formas de minimizar tal dificuldade de análise dos dados obtidos.

O trabalho de Einsfeld *et al.* (2008) apresenta um *framework* que busca integrar a visualização de informação a ambientes de realidade virtual utilizando representações gráficas em três dimensões, com o intuito de gerar visualizações menos abstratas. O *framework* utiliza ainda uma ontologia, o que permite a integração de informações semanticamente relacionadas diretamente na imagem. Os autores argumentam que o conceito proposto ajuda usuários iniciantes a interagir intuitivamente com as visualizações e compreender os dados mostrados.

No trabalho de Huang *et al.* (2010), os autores desenvolvem padrões de projeto para o uso de visualização de informação em sistemas móveis de informações sobre saúde. É feita uma discussão sobre a importância da visualização de informação nesses sistemas e sobre a necessidade de uma aplicação sistemática de padrões de projeto para implementar a visualização em tais sistemas. Eles dissertam, então, sobre a forma como padrões de projeto são aplicados em sistemas móveis e propõem um padrão modificado de acordo com requisitos identificados pelos próprios.

De acordo com Lau e Moere (2007), diversos fatores têm facilitado o recente crescimento e importância da visualização de informação. São eles:

- Disponibilidade de aplicações: várias aplicações que têm emergido recentemente se especializam na produção de complexos artefatos visuais. Tais aplicações têm interfaces intuitivas que resultam num processo de criação que

permite que os usuários obtenham representações gráficas de forma direta e iterativa, sem que precisem compreender questões complexas de configuração. Algumas delas são apoiadas por comunidades virtuais que encorajam o compartilhamento e a criatividade.

- Disponibilidade de dados: a Internet tem facilitado a criação, coleta e compartilhamento de dados de todos os tipos.
- Velocidade e distribuição da Internet: as altas velocidades disponíveis para conexão à Internet têm possibilitado que dados se tornem cada vez mais acessíveis. Alem disso, essa acessibilidade não é limitada a dados crus, pois novas interfaces têm sido criadas para o acesso interativo a grandes conjuntos de informação.
- Técnicas interdisciplinares: estudantes de design são cada vez mais expostos a conhecimento interdisciplinar, como técnicas de programação e desenvolvimento de interfaces, suplementando a experiência de design criativo com habilidades na área da computação. Com isso, aparece um grupo de projetistas de visualizações que desejam cruzar as fronteiras entre as duas áreas, inventando, projetando e prototipando novas técnicas.

Devido a esse crescimento, pesquisas na área têm se intensificado não apenas no sentido de aplicar técnicas de visualização de informação a diferentes domínios, mas também buscando aprimorar as técnicas e metodologias já utilizadas.

No trabalho de Wohlfart *et al*. (2008), os autores fazem uma comparação entre ferramentas que trabalham com a visualização de dados temporais. É feita uma discussão sobre a importância do tempo como uma das dimensões dos dados e as distintas características de tal dimensão, e o fato de tais características não serem comumente contempladas pelas funções de ferramentas de visualização. Os autores concluem, após a análise de diversas ferramentas, que tais aplicações só realizam uma pequena parte das possibilidades de visualização de dados com essa dimensão. Eles sugerem então novas idéias de pesquisa na área para fornecer melhor apoio à visualização desse tipo de dados.

Já Craft e Cairns (2008) delineam novas direções para pesquisas na área, buscando desenvolver bases teóricas para atividades da visualização de informação. Os autores argumentam que, para pesquisadores de outras áreas que desejam utilizar técnicas de visualização de informação para interpretar seus dados, não há formas bem conhecidas que os permitam melhorar sua compreensão de seus dados. Isso leva novos pesquisadores da área a buscar trabalhos e técnicas que não se aplicam bem a seus dados ou propósitos específicos. Os

autores identificam algumas causas de tal problema e sugerem métodos de outras áreas de pesquisa para superar essas deficiências.

No trabalho de Liu *et al.* (2008) é discutida a utilização de um *framework* de cognição distribuída em ferramentas de visualização de informação, buscando novas formas de abordar pesquisas na área. Os autores destacam conceitos fundamentais e construções teóricas da abordagem da cognição distribuída com base na literatura da área, e discutem como tal abordagem pode impactar as direções e metodologias de pesquisa na área da visualização de informação.

Em seu trabalho, Filonik e Baur (2009) abordam a estética das representações gráficas geradas por ferramentas de visualização de informação. Os autores discutem os principais problemas relacionados à estética no processo de visualização e relacionam os conceitos de estética e usabilidade, buscando compreender como a estética pode melhorar a usabilidade das visualizações.

Lam *et al.* (2012) procuram propor uma nova forma de avaliar os resultados de técnicas de visualização de informação. Ao analisar diversos aspectos pertinentes às ferramentas de visualização de informação, como a análise dos dados, a experiência do usuário, os algoritmos de visualização utilizados, entre outros, os autores encapsulam as práticas atuais na comunidade de pesquisa na área e fornecem uma abordagem diferente à tomada de decisões sobre qual pode ser a forma mais eficaz de avaliar a eficácia de uma determinada representação gráfica.

2.3.2 Visualização de Software

Compreender software tem se tornado um significativo desafio mesmo para programadores experientes. Esse problema acontece parcialmente devido ao crescente tamanho e complexidade dos programas atuais, seja em termos de código estático (linhas de código), comportamento em tempo de execução ou utilização de memória. Outro problema é que aplicações modernas, como aplicações de comércio eletrônico e plataformas de computação em nuvem, são construídas utilizando componentes de software reutilizáveis, desde simples classes a grandes *frameworks*. Essas características fazem com que a tarefa de obter uma compreensão global do comportamento e estado de um programa torne-se muito desafiadora para qualquer programador (KELLEY *et al.*, 2012).

O tamanho e a complexidade do software são, também, grandes impedimentos para as ferramentas que buscam auxiliar a compreensão de um software, particularmente aquelas que se baseiam em análise estática de código. As técnicas de programação citadas anteriormente fazem com que a utilização de tais ferramentas resultem em informações imprecisas que são de pouco valor para o desenvolvedor. Já ferramentas que analisam o comportamento dinâmico de programas têm tradicionalmente se concentrado na identificação de problemas de desempenho, e não na compreensão geral do programa. A técnica mais comum atualmente disponível para inspecionar o estado de um programa é o depurador, que só costuma ser capaz de auxiliar usuários na compreensão das menores estruturas de dados (KELLEY *et al*., 2012).

Uma subárea mais específica da visualização de informação é a visualização de software. A visualização de software combina técnicas de áreas diferentes como engenharia de software, mineração de dados, computação gráfica e interface homem-máquina, e consiste em criar uma imagem do software por meio de objetos visuais (WARE, 2012; TEYSEYRE e CAMPO, 2009). Esses objetos visuais podem representar, por exemplo, sistemas ou componentes ou seu comportamento em tempo de execução. Como resultado, desenvolvedores podem obter uma percepção inicial de como o software é estruturado, entender sua lógica e explicar e comunicar seu desenvolvimento. Representações gráficas eficazes podem fornecer uma correspondência mais próxima do modelo mental dos usuários do que representações textuais, e tirar vantagem de suas capacidades de percepção (TEYSEYRE e CAMPO, 2009). Isto é, o objetivo da visualização de software não é criar imagens impressionantes, mas sim representações gráficas que evoquem imagens mentais que auxiliem o usuário a obter uma melhor compreensão do software (WARE, 2012).

A visualização de software pode ser vista como a aplicação de técnicas de visualização de informação a softwares, já que os dados coletados de todas as áreas do desenvolvimento de um sistema, como código, documentação e estudos sobre seu uso são abstratos e, portanto, não têm estrutura física associada (GALLAGHER *et al.*, 2008).

A idéia essencial da visualização de software é que representações visuais podem ajudar a fazer com que a compreensão do software seja mais fácil. Imagens do software podem ajudar a diminuir o enfraquecimento do conhecimento, ao ajudar os membros de um projeto a lembrar, e novos membros a aprender, como uma aplicação funciona (BALL; EICK, 1996).

A motivação por trás da visualização de software é reduzir o custo de desenvolvimento e evolução do software. Além de auxiliar desenvolvedores, a visualização pode apoiar a evolução do software ao facilitar a compreensão de tomadores de decisão em

diferentes níveis de abstração e diferentes pontos do ciclo de vida do software (GALLAGHER *et al.*, 2008).

De acordo com Ball e Eick (1996), três propriedades básicas do software podem ser visualizadas:

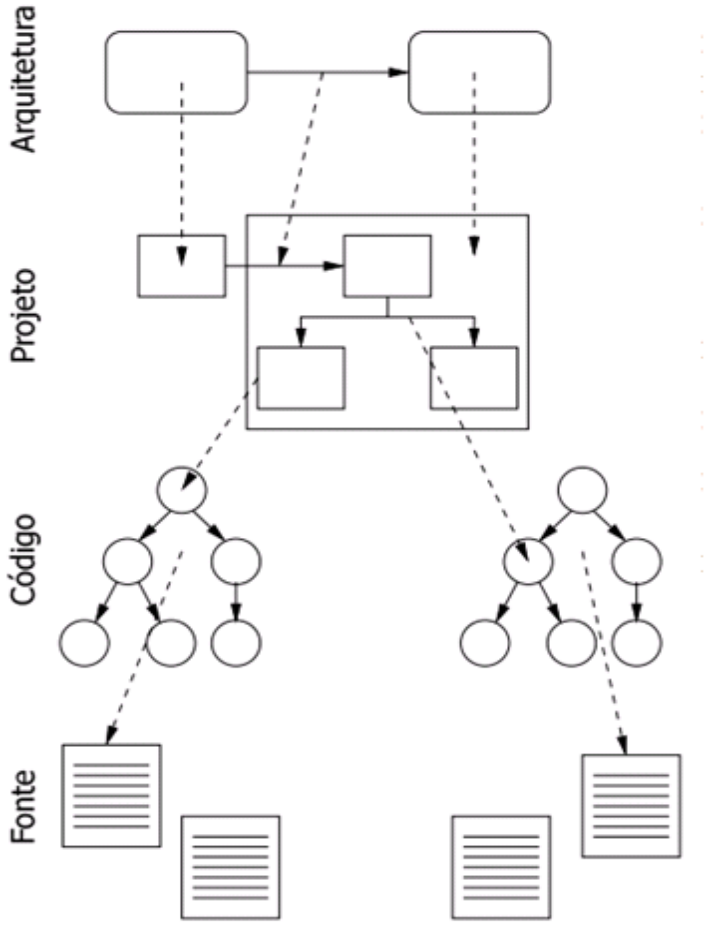
- Estrutura: grafos direcionados são o método mais comum de representar relacionamentos entre entidades de um software. Por exemplo, um vértice pode representar um método e uma aresta pode representar um relacionamento de chamada entre dois métodos.
- Comportamento em Tempo de Execução: animações de algoritmos usam representações de estruturas de dados e movimentos para ilustrar o comportamento de alto nível de algoritmos. Visões de baixo nível podem revelar *bugs* e anomalias de desempenho.
- Código: editores de código que recuam o código e usam fontes ou cores diferentes para diferenciar palavras reservadas são uma forma básica e muito popular de visualização.

De acordo com Pinzger (2005), as visões da implementação de um software podem ser vistas ainda em diferentes níveis de abstração, de acordo com o que se deseja representar. A Figura 9 mostra os diferentes níveis de abstração, as entidades usadas para criar as visualizações, e a hierarquia entre os níveis.

Os quatro níveis de abstração são:

- Arquitetura: no topo estão as visualizações arquiteturais que descrevem os aspectos arquiteturalmente relevantes do sistema. As entidades deste nível de abstração representam módulos de um sistema.
- Projeto: as visualizações de projeto capturam os aspectos de modelagem dos sistemas de software, como suas características de orientação a objeto. As entidades desde nível podem representar diretórios, arquivos, pacotes, classes, objetos, métodos, funções, atributos ou variáveis.
- Código: a visão de código dá uma representação detalhada da estrutura sintática das classes. As entidades do nível representam construções de uma linguagem de programação, como estruturas de seleção e repetição.
- Fonte: a implementação do sistema consiste em um conjunto de documentos, como arquivos de código, arquivos de configuração, comandos de construção, arquivos de *log*, etc.

Figura 9 – Níveis de Abstração.



Fonte: PINZGER (2005).

A abordagem Reflector – proposta no presente trabalho – e a ferramenta homônima que a implementa, descritas no próximo capítulo, buscam representar a estrutura e o comportamento do software em tempo de execução. A abordagem propõe a análise das aplicações através de seus objetos, que se tornam entidades nas visualizações geradas. Portanto, pode-se afirmar que as visões de implementação geradas pelo Reflector se encontram no nível de abstração ‘Projeto’.

3 REFLECTOR

Tendo em vista todas as vantagens do uso de visualizações de software para processos de desenvolvimento, foi criada uma ferramenta que se propõe a analisar a memória utilizada por aplicações Java e representá-la visualmente. Isto é, neste caso, a visualização utiliza dados em tempo de execução para representar visualmente a memória em tempo de execução. Para isso, foi utilizada a API Reflection do Java, que permite o acesso à memória utilizada por uma aplicação em tempo de execução, possibilitando: extrair informações sobre o uso da memória por uma aplicação em qualquer ponto de sua execução; determinar a classe de um objeto; acessar os atributos de um objeto, assim como obter seus valores. A ferramenta fornece ao usuário uma representação visual em forma de grafo de todas as estruturas e variáveis acessíveis a partir de um determinado objeto e os relacionamentos entre tais estruturas e variáveis em um determinado ponto da execução da aplicação. Assim, é possível visualizar toda a memória utilizada pelo programa que é, em um determinado ponto, acessível a partir de um determinado objeto. Tal funcionalidade permite que um desenvolvedor detecte visualmente várias características da aplicação que podem não ser facilmente detectadas quando apenas o código-fonte é analisado. Ela pode ser útil também para a compreensão dos relacionamentos entre objetos da aplicação.

3.1 TRABALHOS RELACIONADOS

Representações gráficas de softwares há muito tempo são bem aceitas como um auxílio à compreensão do desenvolvimento e funcionamento dos mesmos. De acordo com Grundy e Hosking (2000), o interesse por tais ferramentas aparece principalmente no desenvolvimento de grandes sistemas de software. Para auxiliá-los no projeto e na compreensão do software, desenvolvedores buscam apoio visual para a modelagem de aplicações nessas ferramentas. Isso se reflete no número de pesquisas conduzidas no sentido de utilizar representações gráficas de forma eficiente na comunicação de diversas informações relacionadas a softwares.

Feijs e De Jong (1998), por exemplo, criaram a ferramenta ArchView, que utiliza técnicas de extração, visualização e cálculo para analisar a arquitetura de um software. Ela produz uma visualização da arquitetura que apresenta relações de uso entre os componentes

de um sistema. Tais relações são armazenadas em um conjunto de arquivos que é então interpretado, e gera um arquivo VRML que fornece uma representação em 3D da arquitetura do sistema analisado.

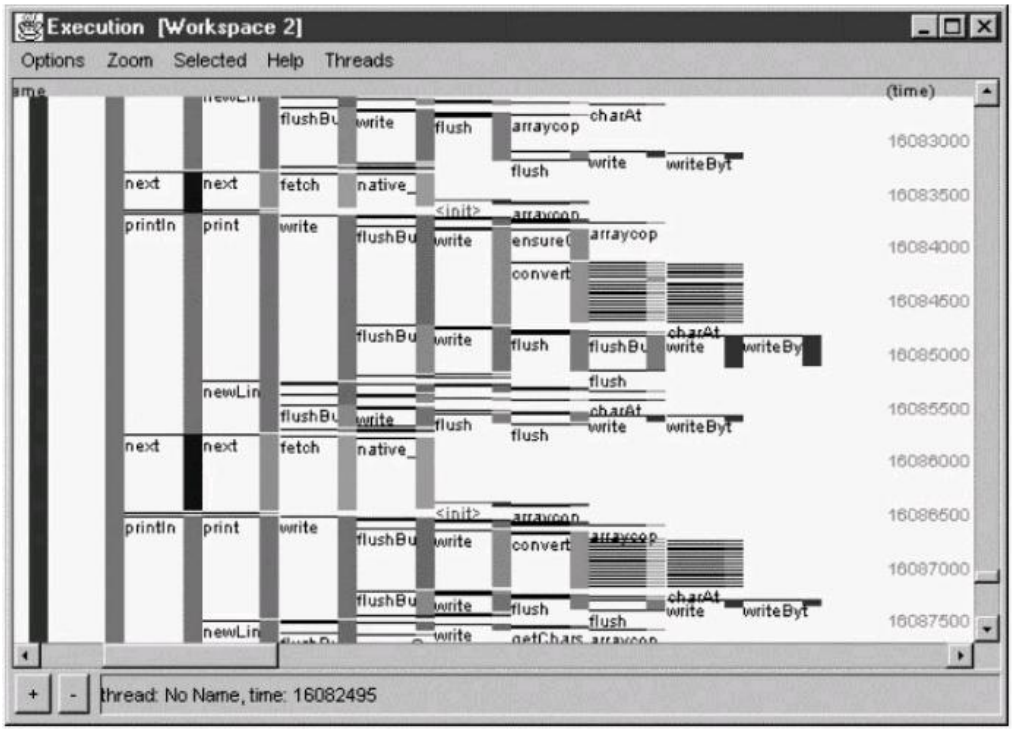
Sim *et al.* (1999) procuram, com a ferramenta *The Searchable Bookshelf*, combinar abordagens de busca e exploração à compreensão de um software. A aplicação permite que usuários explorem a estrutura de um software a partir de uma visão geral inicial ao navegar por uma representação em HTML e uma visualização central do software.

Grundy e Hosking (2000) propõem uma ferramenta chamada *SoftArch*, que é um sistema não apenas de visualização, mas também de modelagem de software, permitindo que informações de sistemas sejam visualizadas em visões arquiteturais. A ferramenta permite a visualização estática e dinâmica de componentes da arquitetura do software e fornece também vários níveis diferentes de abstração.

A linguagem formal LePUS, apresentada por Eden (2002), é uma linguagem dedicada à especificação de arquiteturas orientadas a objeto. Ela é capaz de gerar diagramas que podem ser usados na especificação de padrões de projeto e arquiteturas e na documentação de *frameworks* e programas. Como linguagem visual, a LePUS não busca extrair informações arquiteturais, mas apenas representar a arquitetura de um software de forma simples para comunicação da mesma com tomadores de decisão.

De Pauw *et al.* (2002) desenvolveram a ferramenta de visualização de software *Jinsight.* Similar ao Reflector, a ferramenta procura analisar o uso da memória por aplicações Java em tempo de execução. Sua abordagem se baseia em visões, que são diferentes telas capazes de mostrar diversas informações sobre a utilização da memória da Máquina Virtual Java (JVM). Cada visão fornece um tipo diferente de informação. A visão de histogramas, por exemplo, informa o usuário sobre diversos aspectos relacionados às classes da aplicação, como o tempo gasto por métodos de uma classe, o número de chamadas de métodos, a quantidade de memória consumida ou o número de *threads* nos quais uma classe ou instância participa. Outras visões fornecem informações sobre padrões de referências entre objetos, vazamentos de memória, desempenho, árvores de chamada de métodos, entre outros dados relevantes à execução de uma aplicação. A Figura 10 mostra a visualização gerada pela ferramenta da execução de uma aplicação Java. O comprimento de cada faixa reflete o tempo gasto por cada método, o número de faixas corresponde ao número de métodos chamados na pilha da aplicação, e as cores denotam classes. Cada conjunto de faixas corresponde a uma *thread* da aplicação, e tem como objetivo caracterizar o comportamento da *thread* em questão.

Figura 10 – Um dos diferentes tipos de visualização possíveis na ferramenta *Jinsight*.

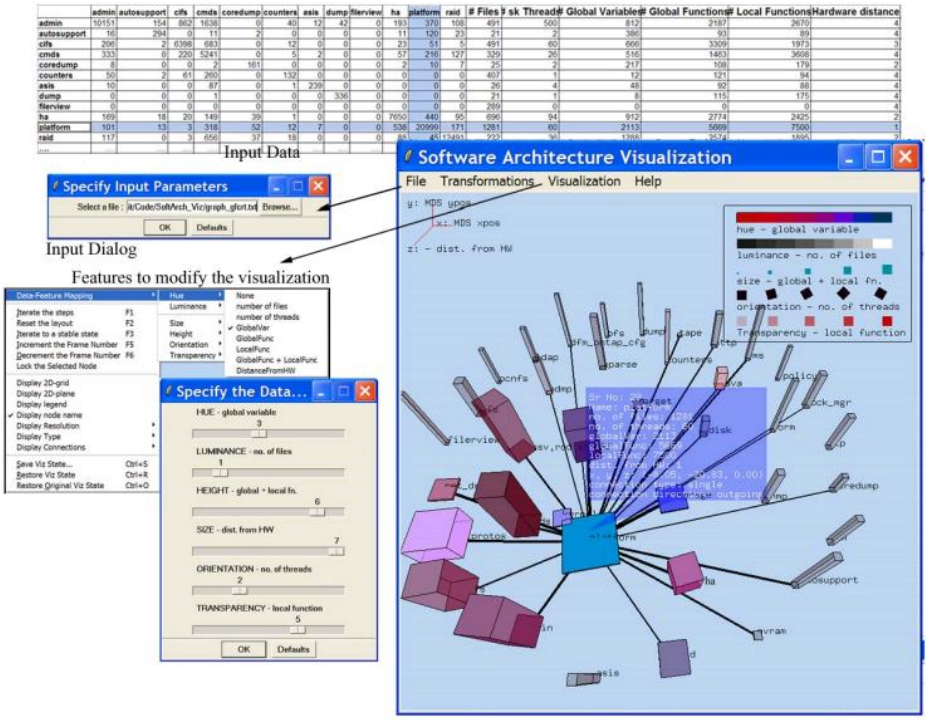


Fonte: DE PAUW *et al.* (2002).

O trabalho de Ducasse e Lanza (2003) utiliza técnicas de visualização de software para representar métricas e apresentar diversos tipos de informação visualmente. A abordagem *Polymetric Views*, proposta pelos autores, cria representações gráficas que incluem diversas informações sobre o software analisado como, por exemplo, sua estrutura hierárquica de classes, o tamanho de suas classes e métodos e o uso de seus atributos. Buscando analisar a qualidade e a complexidade do software, algumas métricas são também inclusas nas imagens geradas. As métricas utilizadas para cada classe são o número de métodos sobrescritos, o número de atributos, o número de métodos, o número de classes estendidas, o número de comandos no corpo dos métodos, entre outras.

Sawant e Bali (2007) apresentam uma ferramenta chamada *SoftArchViz* que utiliza técnicas de escalonamento multidimensional para gerar visualizações da arquitetura de sistemas de software grandes. A ferramenta cria representações de componentes individuais de software que modificam suas propriedades de posição espacial, cor e textura de acordo com o valor de atributos dos componentes. Ela é capaz, ainda, de representar a correspondência na arquitetura em diferentes estágios do ciclo de desenvolvimento do software e, assim, identificar todos os componentes que foram modificados e o tamanho de tais mudanças. A Figura 11 mostra uma imagem do sistema.

Figura 11 – *SoftArchViz*, um sistema de visualização da arquitetura de softwares.

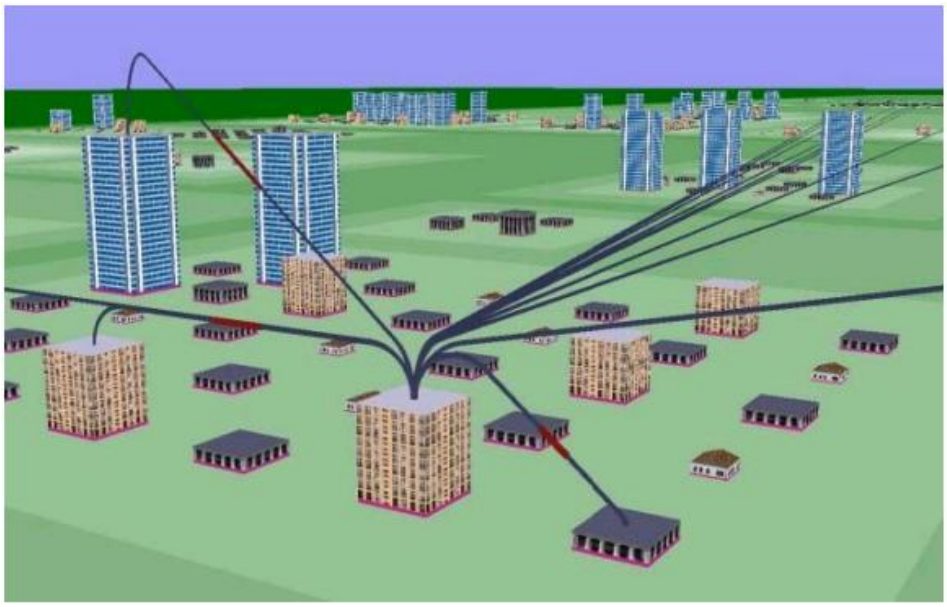


Fonte: SAWANT e BALI (2007).

O trabalho de Dugerdil e Alam (2008) apresenta o sistema *EvoSpaces* de visualização 3D. Com a ferramenta, os autores propõem uma forma de lidar com grandes quantidades de dados em uma visualização. Para isso, o sistema *EvoSpaces* representa a arquitetura do software analisado na forma de uma cidade moderna, com prédios e bairros, em um espaço 3D. A ferramenta representa classes e arquivos como prédios e os relacionamentos entre eles como canos entre os prédios. Os prédios podem pertencer a diferentes categorias de acordo com as métricas utilizadas em uma execução. A Figura 12 mostra um exemplo do tipo de visualização gerada pela ferramenta.

A ferramenta SAVE (*Software Architecture Visualization and Evaluation*), apresentada por Duszynski *et al*. (2009), analisa e otimiza a arquitetura de sistemas de software. Os autores argumentam que a ferramenta implementada, que promove a checagem da equivalência de funcionalidades implementadas em relação a especificações de seu projeto, é capaz de auxiliar desenvolvedores durante processos de desenvolvimento e evolução de sistemas de software.

Figura 12 – Exemplo de visualização de software gerado pela ferramenta *EvoSpaces*.

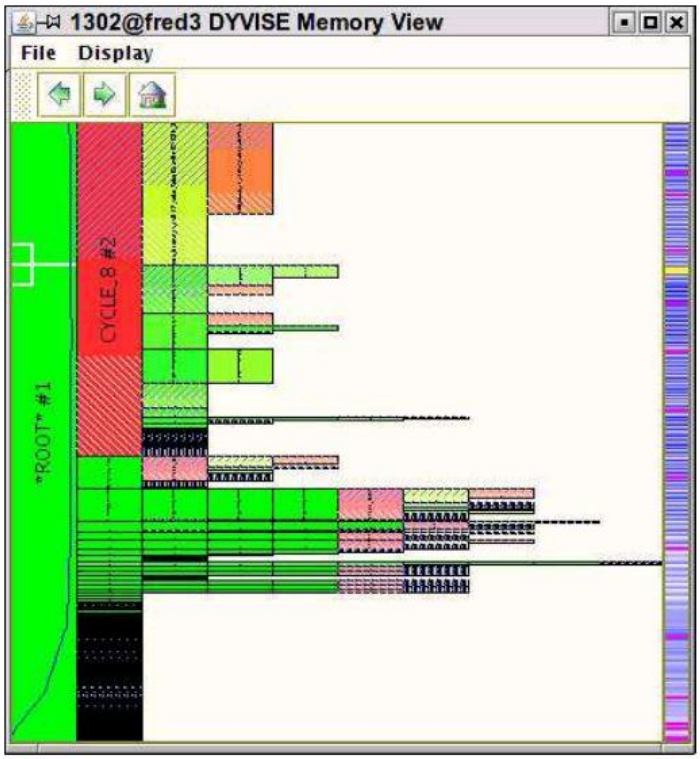


Fonte: DUGERDIL e ALAM (2008).

Reiss (2010) apresenta contribuições que, assim como aquelas da proposta do Reflector, têm base na análise da memória utilizada por aplicações Java. Ao analisar toda a memória de uma Máquina Virtual Java em execução, a ferramenta DYMEM, apresentada pelo autor em seu trabalho, busca monitorar todos os programas Java ativos em um determinado momento na máquina virtual. A partir desse monitoramento é gerada uma visualização compacta da memória, com o objetivo de ajudar desenvolvedores a identificar problemas na utilização da memória por parte dos programas, como falhas na coleta de objetos desnecessários e vazamentos de memória. A Figura 13, retirada do trabalho, mostra um exemplo das visualizações criadas pela ferramenta. O tamanho e a cor dos retângulos são definidos pelo número de objetos existentes de cada classe em um determinado momento e pela quantidade de memória que utilizam.

O trabalho de Myers e Duke (2010) traz uma abordagem que procura gerar visualizações de grafos de memória utilizando elementos visuais que representem grandes estruturas utilizadas no programa, buscando simplificar as imagens criadas. Para isso, os autores propõem a utilização de marcos ou *landmarks*, elementos visuais que são usados para representar certas abstrações de projeto como padrões de projeto e arquiteturas comuns usadas para modelar o estado estático de um software. Os autores argumentam que tais abstrações podem fornecer conexões úteis para a compreensão da estrutura da execução de um software a partir de sua análise em uma representação gráfica.

Figura 13 – Análise de Memória com a ferramenta DYMEM.



Fonte: Reiss (2010).

Como demonstrado nessa subseção, muitos esforços têm sido feitos no sentido de desenvolver ferramentas que auxiliem a análise e compreensão de programas, com a utilização de um número de técnicas para visualizar o comportamento dos mesmos. Parte da pesquisa feita tem tido como objetivo ajudar desenvolvedores a navegar e visualizar código-fonte. Mais útil, porém, é a compreensão das estruturas de dados de um programa. Técnicas de determinação de estruturas de dados dividem-se em duas categorias principais: análise estática e análise dinâmica. Algoritmos de análise estática constróem aproximações de possíveis configurações de estruturas em tempo de compilação. Essas aproximações, porém, são muito imprecisas para tarefas de depuração e compreensão mais detalhadas.

A abordagem proposta e implementada no presente trabalho se relaciona melhor com ferramentas de análise dinâmica, analisando a memória utilizada em execuções reais para fornecer ao usuário um grafo do estado real do programa. Como o maior desafio para tal categoria de ferramentas é a forma como lidam com grandes volumes de dados, a característica que as distingue é a forma como agregam a informação para o usuário. O principal objetivo da abordagem Reflector é ajudar programadores a compreender a organização e estrutura geral das aplicações, então essa agregação é feita levando em

consideração a manutenção mais próxima possível das propriedades estruturais da implementação.

Várias ferramentas já existentes, que também trabalham com análise e visualização de memória de aplicações Java, se preocupam principalmente com a detecção de problemas de desempenho, como vazamentos de memória. A principal diferença entre tais ferramentas e o Reflector é que elas, ao tentarem simplificar a visualização para o usuário, negligenciam detalhes importantes para a compreensão de como as estruturas de dados funcionam. Já o Reflector procura mostrar toda a estrutura do programa de forma fiel à forma real implementada pelo desenvolvedor.

Outra diferença importante é a presença de opções de seleção de quais partes da memória o usuário deseja visualizar e analisar. A abordagem Reflector fornece opções de seleção de classes individuais, seleção de atributos de classes e simplificação de estruturas comuns em grafos. Aplicações Java podem, facilmente, fazer uso de dezenas ou centenas de milhares de objetos. Com as opções de seleção, o usuário pode diminuir drasticamente a quantidade de objetos e, conseqüentemente, o volume dos dados que serão considerados para a criação da visualização. Isso tem, como resultado, representações gráficas da memória de complexidade visual reduzida e maior potencial de utilização para auxílio na compreensão do software analisado.

O restante deste capítulo dedica-se à explanação do funcionamento da abordagem Reflector e sua ferramenta homônima. Será apresentado também um guia de sua utilização, explanando a operação de cada funcionalidade a partir da interface gráfica da ferramenta implementada.

3.2 JAVA REFLECTION

A API Java Reflection permite que o desenvolvedor escreva código capaz de executar ações em tempo de execução utilizando apenas descrições estáticas dos objetos envolvidos. É possível, por exemplo, criar objetos, invocar métodos e acessar ou alterar valores de atributos utilizando apenas as definições estáticas de uma classe. Devido a essas funcionalidades, a API é muito utilizada em aplicações que manipulam dados e recursos que podem estar disponíveis em tempo de execução, mas não estão no momento da compilação. Por exemplo: aplicações que utilizam *plug-ins*, aplicações que usam estruturas de dados definidas em arquivos

externos, aplicações que precisam examinar dados que só se tornam disponíveis após a implantação, ferramentas de testes e depuração e aplicações que precisam se comportar de forma diferente de acordo com o sistema operacional sobre o qual estão sendo executadas (LIVSHITS *et al.*, 2005).

Para o desenvolvimento do presente trabalho, foi utilizado um subconjunto das funcionalidades oferecidas pela API; particularmente, as funcionalidades que permitem a inspeção de objetos. Utilizando a API, é feito um procedimento chamado *observação*. O procedimento consiste na obtenção da definição da classe de um objeto e, subsequentemente, sua lista de atributos. O tipo de cada atributo é então avaliado. Caso o atributo seja primário (tipos primitivos ou *String*), seu valor é armazenado. Caso contrário, o atributo faz referência a outro objeto, que é, por sua vez, observado também.

Desta forma, é possível detectar todas as referências que um objeto contém para outros objetos, e repetir o procedimento sistematicamente. Assim, é possível analisar todos os objetos contidos na memória que podem ser atingidos a partir de um determinado objeto. O procedimento é descrito em maiores detalhes na próxima seção.

3.3 GRAFO DE MEMÓRIA

Utilizando a API Java Reflection, foi desenvolvida a ferramenta Reflector. Ela percebe a memória utilizada por uma aplicação como um grafo direcionado, no qual objetos são vistos como vértices do grafo, e as referências não nulas presentes em cada objeto são vistas como as arestas que emanam de um vértice do grafo para outro.

Conforme sugerido na seção anterior, o grafo é obtido a partir de um dos objetos da aplicação, ao qual a operação de observação é originalmente aplicada, analisando o objeto e as referências que ele contém e aplicando recursivamente a operação de observação. A operação resulta em um subgrafo do grafo de memória da aplicação. Tal subgrafo obtido após a execução do procedimento de observação a partir de um objeto X é chamado de grafo de memória associado a X, ou simplesmente G(X). Como a observação pode ser iniciada a partir de qualquer objeto da aplicação, a escolha do objeto define qual porção da memória utilizada pela aplicação é levada em consideração. Assim, apenas uma fatia da memória é observada pela ferramenta, potencialmente facilitando sua compreensão. O processo de seleção de partes menores de grandes volumes de dados para melhor compreensão se chama *slicing*.

3.3.1 Slicing

O termo *slicing* (fatiamento) foi apresentado inicialmente por Weiser (1981) como uma técnica de decomposição que extrai de um programa os comandos que são relevantes a uma computação em particular, de forma a diminuir a quantidade de dados analisados para a depuração de código-fonte. A técnica parte da suposição de que, em algumas situações, apenas uma parte do comportamento de um programa é interessante para um desenvolvedor durante tarefas de depuração, integração, manutenção, testes, garantia da qualidade, etc (STEINDL, 2000).

Para que a técnica seja utilizada, o comportamento que se deseja observar deve ser especificado de uma certa forma. Diz-se que se o comportamento de interesse puder ser expressado por um conjunto de variáveis e comandos, tal conjunto é considerado o critério de *slicing* de tal comportamento. Assim, o critério de *slicing* oferece uma janela para observar o comportamento de um programa (WEISER, 1981).

O *slicing* que acontece na abordagem Reflector, porém, difere da definição original de *slicing*. Como a abordagem utiliza dados sobre a memória utilizada por uma aplicação, e não seu código-fonte, o fatiamento ocorre sobre a memória com a operação de observação. Assim, o objeto X a partir do qual uma operação de observação é iniciada corresponde ao critério de *slicing*, já que define que porção da memória será observado. Portanto, o resultado do processo de extração do grafo de memória a partir de um objeto revela não toda a memória utilizada por uma aplicação, mas apenas a porção da memória alcançável a partir de tal objeto.

3.3.2 Extração do Grafo de Memória

A partir de um objeto X fornecido pelo usuário, o Reflector executa uma busca em largura procurando atingir todos os objetos que são acessíveis a partir de X. Para cada objeto não-primário encontrado, uma instância da classe ReflectorObject é criada. Ela armazena uma referência ao objeto, o valor de cada um de seus atributos no momento da busca e uma identificação exclusiva. Uma representação do algoritmo em pseudocódigo é mostrada na Figura 14.

Figura 14 – Algoritmo utilizado pelo Reflector para observar a memória de uma aplicação.

ALGORITMO `ReflectorCrawler`	
ENTRADA:	Objeto X fornecido pelo usuário.
SAÍDA:	G(X).

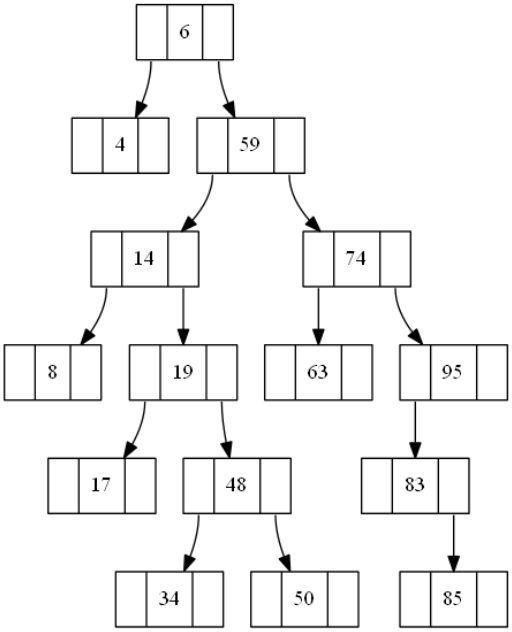
```
1.  Criar uma lista encadeada L e uma árvore binária balanceada de busca T
2.  Inserir X em L
3.  enquanto L não estiver vazia faça
4.      Y ← Remover primeiro objeto de L
5.      se Y ∉ T então
6.          Criar instância de ReflectorObject com Y and inseri-la em T
7.      Buscar ReflectorObject de Y em T
8.      Buscar a definição da classe de Y
9.      para cada campo F da classe
10.         se F for de um tipo primário então
11.             Salvar o conteúdo de F no ReflectorObject de Y
12.         senão
13.             se F referenciar um objeto R ∉ T então
14.                 Inserir R em L
15.             Adicionar relação ao ReflectorObject de Y
16.     fim
17. fim
```

Fonte: Autoria própria.

Perceba que G(X) é conectado e pode conter ciclos. É possível, portanto, que o Reflector encontre o mesmo objeto mais de uma vez ao executar a busca. Para evitar a criação de referências duplicadas, cada objeto deve ser atribuído a apenas uma instância da classe ReflectorObject e vice-versa. Para isso, cada instância de ReflectorObject é armazenado em uma árvore binária de busca balanceada. A árvore é indexada pelo código *hash* do objeto ao qual cada ReflectorObject se refere. Assim, é possível buscar o ReflectorObject que contém a referência a um determinado objeto em tempo O (log_2 (m)), onde m é o número de objetos observados pelo Reflector no momento da busca.

Depois que a busca é finalizada, informações sobre todos os objetos em G(X) foram coletadas e armazenadas. Com tais informações, o Reflector é capaz de gerar automaticamente uma visualização que representa a memória usada pela aplicação e que é acessível a partir de X. Tal representação visual representa aquela parte da memória no momento em que a ferramenta é utilizada. A Figura 15 mostra o resultado gerado quando o Reflector é aplicado a uma árvore binária de busca. Note que cada objeto na estrutura tem uma chave e duas referências a objetos da mesma classe. Nesse caso, a Figura 15 representa o grafo de memória associado ao objeto indexado pela chave de número 6: todos os objetos que podem ser acessados a partir das referências desse objeto.

Figura 15 – Árvore binária de busca.



Fonte: Autoria própria.

3.3.3 Filtro de Seleção

O usuário pode limitar a busca se desejar visualizar apenas um subconjunto dos objetos da aplicação. Essa limitação pode ser feita de duas formas diferentes: através da filtragem de classes e através da filtragem de atributos. O filtro de classes permite que o usuário declare quais classes devem ser consideradas na busca. Apenas objetos que pertencem às classes declaradas pelo usuário serão então observados pela ferramenta. Vale salientar que isso trunca a busca: até objetos de classes especificadas pelo usuário podem não ser observados caso só sejam acessíveis através de cadeias de referências que passam por objetos de classes que não foram declaradas..

O filtro de atributos permite que o usuário especifique, para cada classe observável na aplicação, quais atributos devem ser inspecionados. Isso permite que o Reflector ignore todos os outros atributos da classe que não foram selecionados pelo usuário e dá ao usuário a habilidade de controlar a ordem em que os atributos aparecem na visualização. Assim, o usuário pode personalizar o nível de detalhamento da visualização, diminuindo sua complexidade.

Quando o filtro de seleção é usado, o algoritmo que o Reflector executa para coletar toda a informação sobre o grafo de memória da aplicação é levemente diferente. A Figura 16 ilustra o algoritmo. Note que as linhas 9 e 10 não estão presentes no algoritmo original. Essas linhas asseguram que apenas as classes e atributos que foram selecionados pelo usuário serão observados pelo Reflector; todo o resto é ignorado.

Figura 16 – Algoritmo utilizado pelo Reflector com a utilização de filtros de seleção.

ALGORITMO `ReflectorCrawler2`

ENTRADA: Objeto X fornecido pelo usuário e listas L1 e L2 de classes e campos declarados pelos usuários, respectivamente.
SAÍDA: G(X).

```
1.  Criar uma lista encadeada L e uma árvore binária balanceada de busca T
2.  Inserir X em L
3.  enquanto L não estiver vazia faça
4.      Y ← Remover primeiro objeto de L
5.      se Y ∉ T então
6.          Criar instância de ReflectorObject com Y and inseri-la em T
7.      Buscar ReflectorObject de Y em T
8.      Buscar a definição da classe de Y
9.      se a classe ∈ L1 então
10.         para cada campo F ∈ L2
11.             se F for de um tipo primário então
12.                 Salvar o conteúdo de F no ReflectorObject
13.             senão
14.                 se F referenciar um objeto R ∉ T então
15.                     Inserir R em L
16.             Adicionar relação ao ReflectorObject de Y
17.     fim
18. fim
```

Fonte: Autoria própria.

3.4 PROCESSO DE CRIAÇÃO

Nesta seção, será descrito o processo de criação de visualizações utilizado pela abordagem Reflector e sua ferramenta homônima, de acordo com as definições apresentadas na subseção 2.3.1.

3.4.1 Preprocessamento e Transformação dos Dados

Como mencionado na seção 3.2, os dados sobre o conteúdo de uma aplicação executada na memória de uma Máquina Virtual Java podem ser obtidos com o uso da API Java Reflection. Tais dados, porém, precisam ser estruturados para o uso pela ferramenta Reflector. Durante o processo de obtenção, descrito em detalhes na seção anterior, os dados referentes ao conteúdo de cada objeto não-primário da aplicação sendo analisada são organizados em instâncias da classe ReflectorObject. Cada instância da classe refere-se ao conteúdo de um único objeto da aplicação, e organiza os dados de tal objeto de forma estruturada. Cada instância contém uma referência ao objeto cujos dados armazena, um nome único, um tipo, duas tabelas *hash* que guardam o conteúdo dos campos do objeto – uma para dados de tipos primários e outra para referências a outros objetos da aplicação – e uma lista de adjacência com os nomes únicos de todos os vizinhos no objeto.

3.4.2 Mapeamento Visual

Para a escolha de como efetuar o mapeamento visual para a criação de representações gráficas na implementação da abordagem Reflector, foram analisadas duas ferramentas de desenho de grafos: a ferramenta DOT (GANSNER *et al.*, 1993) e o *toolkit* de visualização Prefuse (HEER *et al.*, 2005).

Apesar de mais recente que a DOT, a ferramenta Prefuse não apresenta algumas das opções de customização e detalhamento de vértices que a anterior fornece. Tais opções incluem a divisão de vértices em múltiplos espaços e o controle específico do posicionamento das arestas em relação à posição dos vértices, características importantes para a implementação da abordagem Reflector.

Já a ferramenta DOT fornece várias opções de definição manual de elementos gráficos no momento da criação de uma representação gráfica. Através dos diversos tipos de elementos disponíveis, é possível definir um tipo específico para cada vértice de um grafo, customizar cada vértice com diferentes rótulos, cores, separação em quantas partes forem necessárias, e controle independente do posicionamento das arestas que saem de cada vértice.

Por tais motivos, o motor de desenho DOT e sua linguagem homônima foram escolhidos para a criação de representações gráficas na ferramenta Reflector. Como mencionado anteriormente, o motor e sua linguagem fornecem diversas opções de elementos gráficos e customizações, possibilitando um alto nível de detalhamento nas representações gráficas criadas com seu uso. Ela permite ainda a definição automática do substrato espacial pelo motor de desenho de acordo com o número de elementos gráficos definidos para cada exibição e os relacionamentos existentes entre os elementos.

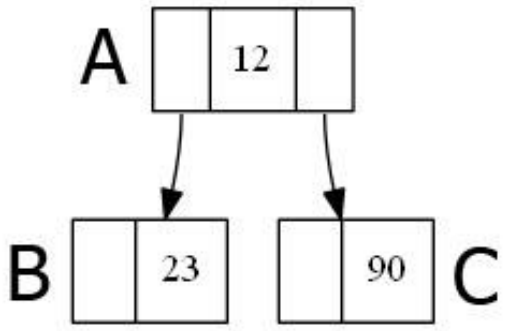
Para as visualizações do presente trabalho, foram utilizadas os seguintes elementos gráficos:

- Objetos: cada objeto de uma aplicação que é analisada pela ferramenta corresponde a um vértice do grafo de memória. Assim, cada vértice do grafo é representado por uma estrutura do tipo *record*. Através dessa estrutura, a linguagem DOT permite a divisão de um elemento visual básico em seções fisicamente separadas. Na representação gráfica final, cada seção de um vértice corresponde a um campo distinto do objeto representado por tal vértice. Cada seção mostra o dado contido naquele campo do objeto, caso o dado seja de tipo primário. Espaços em branco aparecem em três ocasiões: quando um campo faz referência a outro objeto da aplicação, quando um campo tem valor nulo e quando um campo contém uma String vazia.
- Referências: cada referência existente entre dois objetos da aplicação é mapeada para uma aresta entre os vértices que representam tais objetos na representação gráfica. Cada aresta sai da seção do objeto que representa o campo que guarda a referência e culmina em uma seta apontada para o objeto referenciado.

A Figura 17 mostra um exemplo dos elementos gráficos supracitados. O objeto fictício A, representado por uma estrutura do tipo *record*, tem três campos: seu primeiro campo, representado pela primeira seção do elemento gráfico, faz referência ao objeto B; o segundo campo contém um número inteiro; e o terceiro campo faz referência ao objeto C. As referências entre os objetos são evidenciadas pelas arestas, que partem das seções do objeto A e culminam nos objetos B e C. Os objetos B e C, por outro lado, têm apenas dois campos: o primeiro campo de ambos está vazio, o que significa que contêm referências nulas ou um texto vazio; o segundo campo guarda um número inteiro.

Outros elementos gráficos e certas customizações são utilizados para a representação de estruturas especiais existentes no grafo de memória e algumas opções que estão disponíveis para o usuário. Tais elementos gráficos e seu significado nas representações gráficas geradas pelo Reflector são exemplificados na seção 3.5.

Figura 17 – Exemplo dos elementos gráficos utilizados nas visualizações geradas pela ferramenta Reflector.



Fonte: Autoria própria.

3.4.3 Visões

O resultado final do processo de geração do Reflector pode ser visto em diversos exemplos de grafos de memória nas figuras do presente capítulo. O motor de desenho DOT procura organizar todos os objetos na visualização gerada utilizando o espaço da melhor forma possível, e a quantidade de informação representada pode ser limitada livremente pelo usuário com a seleção de objetos que produzam grafos menos densos ou limitando o escopo do grafo utilizando o filtro descrito na subseção 3.3.3.

3.5 VISUALIZAÇÃO NO REFLECTOR

O Reflector possui algumas opções de customização que permitem que o usuário torne a representação do grafo de memória mais adequada para qualquer propósito específico. Nas

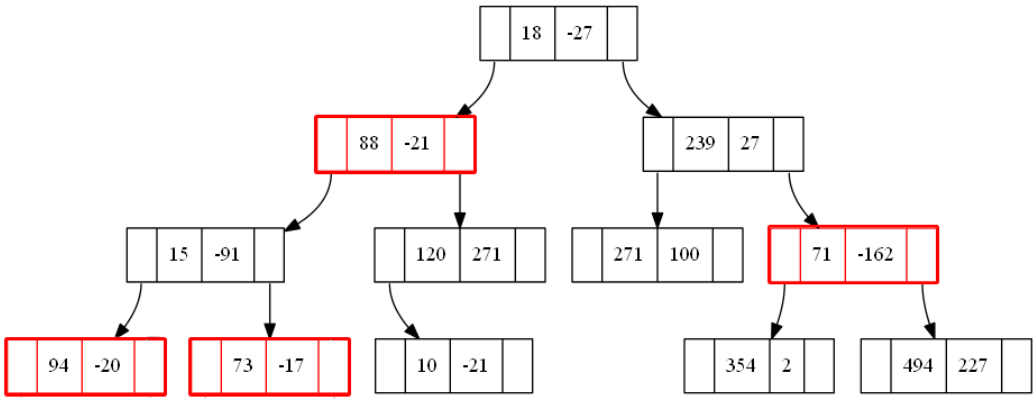
subseções seguintes serão apresentadas as seguintes funcionalidades: os filtros de destaque, as simplificações automáticas e a diferença entre grafos de memória.

3.5.1 Filtro de Destaque

O filtro de destaque é uma funcionalidade através da qual o usuário pode destacar objetos no grafo de acordo com o conteúdo de seus atributos. Cada filtro tem o seguinte formato: (<Att>, <Rel>, <Val>), onde <Att> é o identificador do atributo, <Val> é o valor ao qual o valor do atributo será comparado, e <Rel> é o operador que representa o resultado esperado de tal comparação. Os resultados possíveis são >, == e <.

A Figura 18 mostra o grafo de memória de uma árvore binária que guarda dois atributos, x e y, que são números inteiros. Dois filtros foram aplicados a todos os objetos que compõem a árvore binária: um que filtra objetos cujo atributo x é maior que 50 e outro que filtra objetos cujo atributo y é menor que 0. Isto é: dois filtros estão sendo aplicados simultaneamente: (x, >, 50) e (y, <, 0). Os objetos que atendem a ambos os requisitos são automaticamente destacados em vermelho. Um exemplo de uma potencial utilização para o filtro de destaque é a detecção de objetos que podem contribuir para a ocorrência de um comportamento indesejado em uma aplicação devido à presença de um determinado valor em um dos atributos do objeto.

Figura 18 – Exemplo da aplicação de filtros.



Fonte: Autoria própria.

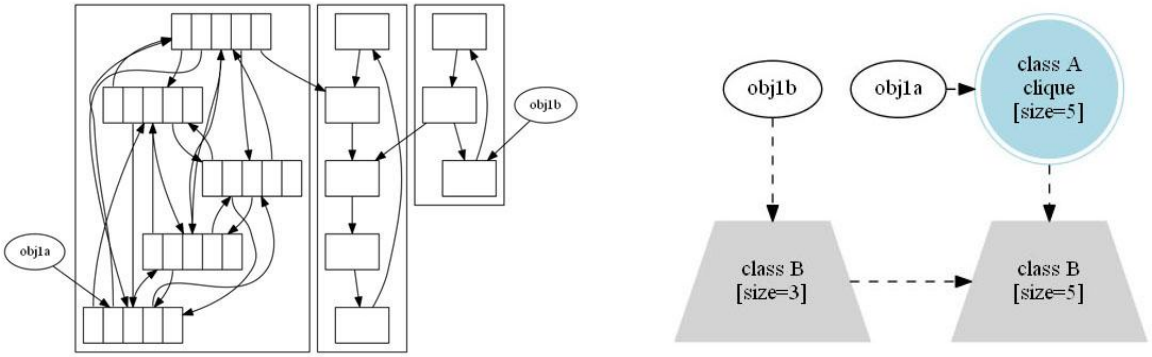
3.5.2 Simplificações Automáticas

Grandes aplicações com centenas de milhares de objetos podem levar a visualizações confusas. Tal confusão pode prejudicar a compreensão de uma representação visual, limitando sua potencial utilidade.

Buscando diminuir tal carga visual, foram utilizados algoritmos específicos para a detecção de subconjuntos de objetos com características de interconexão peculiares. Especificamente, foi utilizado o algoritmo de Bron-Kerbosch (BRON; KERBOSCH, 1973) para detecção de cliques, i.e., a detecção de subconjuntos de vértices de um grafo nos quais cada vértice é adjacente a cada um dos outros vértices. Foi utilizado também o algoritmo de Kosaraju (SHARIR, 1981) para a detecção de componentes fortemente conectados, i.e., grupos de vértices nos quais qualquer vértice é atingível a partir de qualquer outro vértice do subconjunto. Desta forma, é possível representar essas estruturas, notáveis por sua alta conectividade, de forma simplificada nas representações gráficas geradas pelo Reflector.

Na Figura 19 (a), vemos um grafo de memória que contém dois componentes fortemente conectados e um clique. As estruturas foram destacadas em retângulos para facilitar sua visualização. Em um caso real, poderia ser muito mais difícil distinguir tais estruturas. A Figura 19 (b) mostra o mesmo grafo de memória após a execução dos algoritmos de detecção de cliques e componentes fortemente conectados e a aplicação de seus resultados como simplificações no grafo. O círculo azul representa um clique e os trapézios cinza representam componentes fortemente conectados.

Figura 19 – Ilustração de Simplificações Automáticas.



(a) (b)

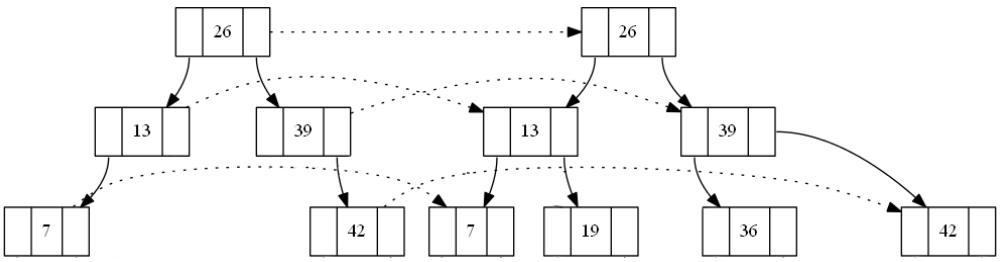
Fonte: Autoria própria.

Outra forma de simplificação proposta pelo Reflector é o agrupamento de objetos. Ela consiste na detecção e destaque automáticos de grupos de objetos que pertencem à mesma classe e são conectados uns aos outros. Ao detectar a ocorrência de um grupo do tipo, tais objetos são agrupados em um subgrafo que é destacado dentro de um retângulo na visualização final gerada. Dois exemplos podem ser vistos na Figura 29. O objetivo dessa forma de simplificação é auxiliar a identificação visual do tipo de estrutura de dados utilizado por uma aplicação. Na Figura 29, por exemplo, é destacada a utilização de uma árvore binária.

3.5.3 Diferença entre Grafos de Memória

A Figura 20 mostra um exemplo do uso da diferença entre grafos de memória. A funcionalidade permite que o usuário observe o grafo de memória da aplicação em diferentes pontos de sua execução. Para isso, o usuário especifica pontos na aplicação nos quais operações de observação do grafo de memória são feitas. Ao visualizar esses grafos de memória lado a lado, é possível perceber como as estruturas de dados e o uso de memória da aplicação mudam com o tempo.

Figura 20 – Diferença entre grafos de memória.



Fonte: Autoria própria.

Para melhor visualização, o Reflector procura conectar visualmente objetos que não mudaram significativamente de um ponto para outro. Isso é feito ao comparar objetos que compartilham o mesmo identificador em dois grafos de memória. Se o método *equals()* retornar o valor *true* quando aplicado a dois objetos com o mesmo identificador, os dois objetos são o mesmo nos dois grafos. Eles são então representados na mesma altura e conectados por uma aresta pontilhada na imagem gerada pelo Reflector. Essa heurística

procura ajudar o usuário, já que a estrutura do grafo pode mudar drasticamente entre dois pontos. Tal mudança pode acontecer devido à criação de novos objetos e/ou novas referências. Ao conectar objetos equivalentes e desenhá-los na mesma altura, torna-se mais fácil para o usuário identificar quais objetos são novos e quais objetos estão nos dois grafos.

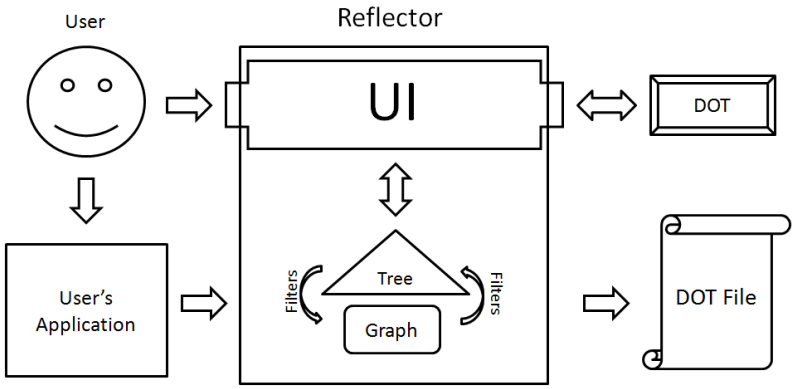
Seria possível, claro, executar uma busca exaustiva nas árvores para cada objeto, o que resultaria em mais equivalências entre objetos, mas isso teria um custo computacional proibitivo para aplicações de médio e grande porte. Ao comparar apenas objetos que compartilham o identificador, é tipicamente encontrado um número razoável de equivalências, o bastante para assegurar que as estruturas dos grafos parecerão razoavelmente similares na imagem final.

Tal heurística permite que os grafos pareçam similares mesmo que suas estruturas tenham mudado, facilitando a detecção, por parte do usuário, de quais objetos mudaram e como eles mudaram. A Figura 20 mostra uma árvore de busca binária em dois pontos: antes e depois da inserção dos elementos 19 e 36. Note que os objetos que estão presentes nos dois grafos são conectados por arestas pontilhadas.

3.6 ARQUITETURA

A Figura 21 mostra a arquitetura do Reflector. Os principais componentes da ferramenta são uma árvore binária de busca que guarda referências a objetos observados, uma estrutura de grafo que armazena os relacionamentos entre esses objetos e uma interface de usuário. A árvore binária é indexada pelos códigos *hash* dos objetos armazenados e pode salvar múltiplos objetos com o mesmo código *hash*. Cada objeto observado pelo Reflector é adicionado à árvore. Depois que todo o grafo de memória foi observado, a árvore é usada para recuperar os valores de todos os campos dos objetos. O grafo é usado para replicar a estrutura formada pelas referências dos objetos. Essa réplica é usada na execução dos algoritmos que possibilitam as simplificações propostas na subseção 3.3.2.

Figura 21 – Arquitetura do Reflector.



Fonte: Autoria própria.

Os filtros e simplificações configurados pelo usuário na interface gráfica afetam os dados armazenados pela ferramenta na árvore e no grafo, determinando a representação gráfica gerada. Como resultado dessas configurações, o Reflector pode gerar dois tipos de saída: a imagem do grafo de memória criada pelo motor de desenho DOT, que pode ser visualizada na interface e salva para visualizações posteriores; e um arquivo na linguagem DOT que especifica o grafo de memória e pode ser utilizado em qualquer ferramenta de visualização que tem suporte a esse tipo de especificação.

3.7 UTILIZAÇÃO DO REFLECTOR

Nesta seção, explanamos a utilização do Reflector posicionando-o no ciclo de desenvolvimento de software e especificando o funcionamento da interface gráfica da ferramenta implementada.

3.7.1 Posicionamento no Ciclo de Desenvolvimento

Conforme mencionado na seção 2.2, um processo de software é um conjunto de atividades e resultados associados que geram um produto de software. Modelos de processo de software organizam as atividades do processo de formas distintas, mas quatro atividades

são comuns a todos: a especificação o software, o desenvolvimento do software, a validação do software e a evolução do software.

A abordagem Reflector tem duas atividades como principais momentos para sua utilização: as atividades de desenvolvimento do software e de evolução do software. A atividade de desenvolvimento pode ser apoiada pela abordagem com o seu uso em tarefas de depuração. Como citado na seção 3.3, o Reflector é capaz de, através de um objeto inicial, descobrir toda a memória utilizada por uma aplicação que pode ser alcançada através de referências desse objeto inicial. Além disso, é possível, com a utilização dos filtros de destaque e seleção, simplificar a representação da memória e destacar objetos que tenham características específicas. Com isso, o processo de depuração de uma aplicação pode se tornar menos dispendioso, custoso e enfadonho, já que os dados que precisam ser avaliados para a descoberta e retirada de erros podem ser refinados e classificados e, assim, a busca por um comportamento que está causando erros pode se tornar mais rápida e fácil.

Já a atividade de evolução, que é o foco principal da utilização do Reflector no presente trabalho, pode ser apoiada pela abordagem principalmente através de seu uso para tarefas de comunicação de conhecimento e documentação. Um problema recorrente na atividade de evolução é que o grupo de pessoas que desenvolve o software é diferente daquele que faz a evolução do software, que pode também ser composto por pessoas diferentes em diversos momentos no tempo. Essa diferença prejudica a compreensão que uma equipe tem sobre o funcionamento de um software e sobre a forma como ele foi modificado desde o momento de sua implantação até um determinado ponto em sua evolução.

Assim, uma das propostas do presente trabalho é a utilização das representações gráficas geradas pela abordagem Reflector para o auxílio da compreensão do funcionamento de uma aplicação por parte de uma equipe de evolução.

3.7.2 Utilização da Interface Gráfica

Para utilizar a ferramenta, o usuário deve incluir o arquivo JAR do Reflector em seu projeto. Ele deve então instanciar a classe Reflector em uma aplicação, fornecendo, através do construtor, a referência através da qual a ferramenta deve começar a observar a memória e um nome que será usado para nomear arquivos temporários criados pela ferramenta. Para iniciar a

interface de usuário do Reflector, o usuário deve invocar o método *loadDisplay*. Um exemplo é mostrado na Figura 22.

Figura 22 – Como utilizar o Reflector.

```
public static void main (String args[]) {
    DemoOne demo = new DemoOne();
    Reflector reflect = new Reflector (demo, "demo");
    reflect.loadDisplay();
```

Fonte: Autoria própria.

Se o usuário desejar usar a funcionalidade de diferença entre grafos de memória, alguma configuração é necessária antes de carregar a interface de usuário. A Figura 23 mostra um exemplo.

Após instanciar a classe Reflector, o usuário deve definir quando as imagens da memória devem ser criadas na aplicação invocando o método *getSnapshot*. Uma imagem da memória é criada e armazenada no momento que o método é invocado. O usuário pode criar quantas imagens desejar. Para mostrar a imagem resultante, contendo todos os grafos, o método *loadDisplay* deve receber um valor *true* como argumento. A imagem gerada ao executar o código mostrado na Figura 23 é mostrada na Figura 20.

Figura 23 – Como utilizar a diferença entre grafos.

```
BinTree tree = new BinTree (26);
Reflector reflect = new Reflector (tree, "tree");
tree.add (13);
tree.add (39);
tree.add (7);
tree.add (42);
reflect.getSnapshot();
tree.add (19);
tree.add (36);
reflect.getSnapshot();
reflect.loadDisplay (true);
```

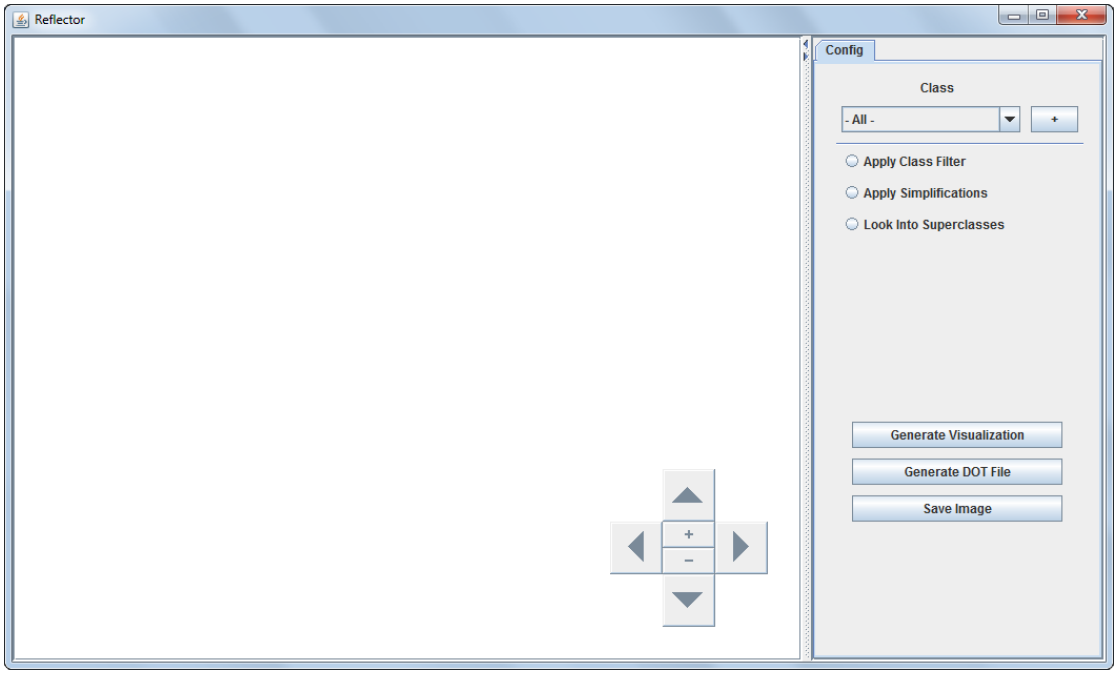
Fonte: Autoria própria.

A tela inicial do Reflector consiste na área onde o grafo de memória será mostrado, uma área com as opções de customização e os botões para gerar as visualizações dos grafos e criar especificações na linguagem DOT. Nessa tela, é possível também especificar classes da

aplicação às quais se deseja aplicar customizações específicas. A tela pode ser vista na Figura 24.

Para configurar filtros para classes individuais, o usuário deve adicionar a classe que deseja filtrar ao menu *drop-down* da tela utilizando o botão + e seguir as orientações mostradas nas janelas de diálogo. A Figura 25 mostra um exemplo. A Figura 25 (a) mostra o diálogo necessário para a definição do pacote ao qual a classe que se deseja adicionar pertence. Com o nome do pacote, a ferramenta Reflector é capaz de listar todas as classes existentes no pacote, como mostra a Figura 25 (b).

Figura 24 – Tela inicial do Reflector.

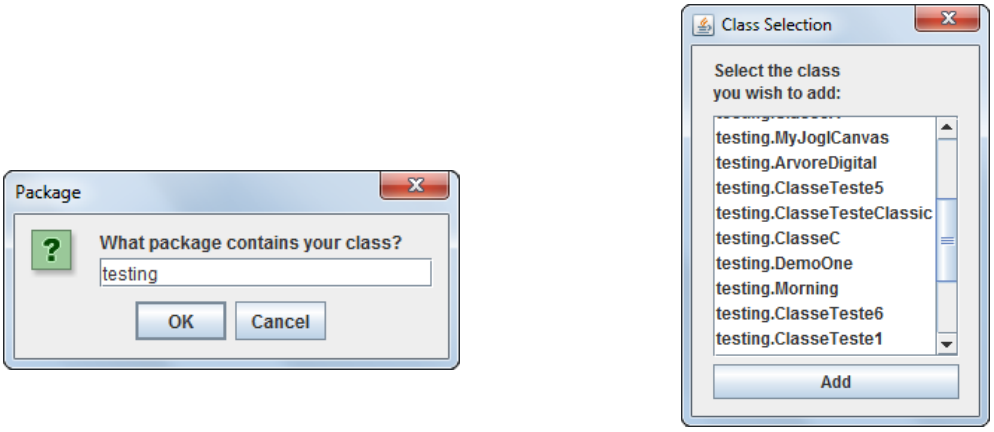


Fonte: Autoria própria.

Essa facilidade, porém, só está disponível para pacotes declarados diretamente na aplicação que está sendo analisada. Classes pertencentes a pacotes nativos do JDK (*Java Development Kit*) instalado na máquina do usuário devem ser declaradas com seu nome completo, incluindo o nome completo do pacote. Neste caso, a ferramenta apresenta uma janela de diálogo diferente, mas semelhante à que aparece na Figura 26 (a), onde o usuário deve inserir o nome da classe. O nome da classe desejada, caso realmente exista no sistema, aparecerá, então, no menu. Caso contrário, a ferramenta informa o usuário que não encontrou a classe desejada, mostrando a janela de diálogo da Figura 26 (b).

A partir do momento que uma classe é adicionada com sucesso ao menu, a filtragem de classes pode ser ativada com a seleção da opção ‘Apply Class Filter’. Assim, o usuário ativa o filtro de classes especificado na subseção 3.3.2. Ainda na tela inicial, a opção ‘Apply Simplifications’ ativa as simplificações automáticas explanadas na subseção 3.5.2. O botão ‘Look Into Superclasses’ indica a opção de considerar ou não campos pertencentes a superclasses das classes dos objetos observados pela ferramenta. Caso a opção seja selecionada, tais campos são considerados pelo Reflector; caso contrário, somente os campos declarados diretamente nas classes dos objetos observados são analisados.

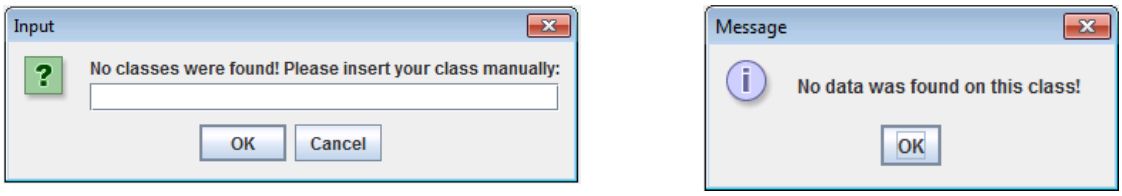
Figura 25 – Janelas de diálogo do Reflector.



(a) (b)

Fonte: Autoria própria.

Figura 26 – Outras janelas de diálogo do Reflector.

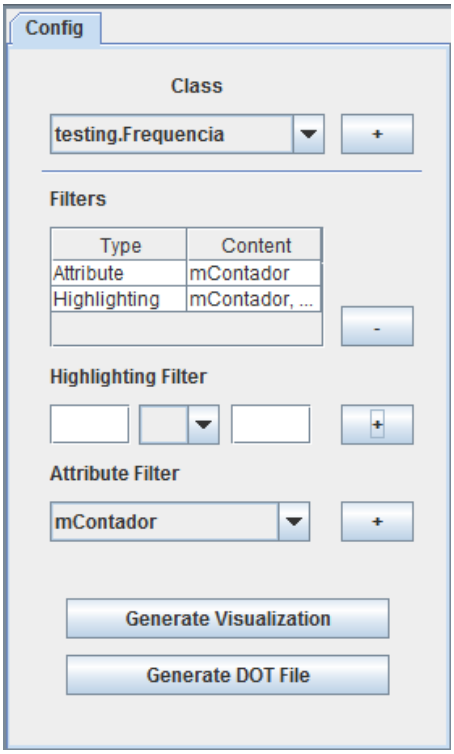


(a) (b)

Fonte: Autoria própria.

Quando o usuário seleciona uma classe no menu, a tela mostra as opções de filtragem de seus atributos. A Figura 27 mostra a tela nesse ponto. Ela contém uma lista de todos os filtros já configurados para aquela classe e fornece opções para a adição de outros filtros. Todos os filtros detalhados na seção 3.3 estão disponíveis.

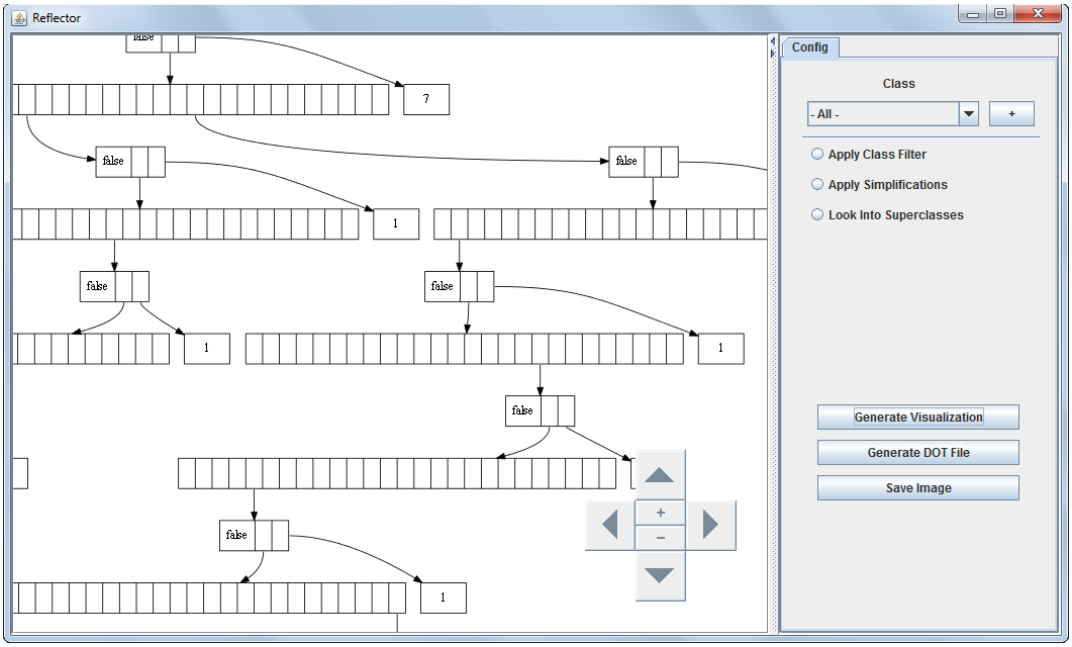
Figura 27 – Opções de configuração de filtros.



Fonte: Autoria própria.

O botão ‘Generate Visualization’ faz com que o Reflector mostre a imagem do grafo de memória. Para isso, ele cria uma especificação temporária do grafo em linguagem DOT, levando em consideração todos os filtros e simplificações configurados pelo usuário. O motor DOT é então usado para gerar a imagem do grafo em formato PNG, que é então mostrada na interface de usuário. A Figura 28 mostra um exemplo. Usando os controles encontrados no canto direito inferior da área da imagem, o usuário pode mover, aproximar e distanciar a imagem. Cada operação de *zoom* causa uma nova chamada ao motor DOT, que cria uma nova imagem com o novo tamanho, ao invés de apenas redimensionar a imagem. Isso é feito para preservar a qualidade da imagem e assegurar que todos os rótulos continuam legíveis. Sempre que uma nova imagem é criada através da operação de *zoom*, ela é salva em um *cache* de imagens. Assim, caso o nível de aproximação ao qual ela corresponde seja selecionado novamente pelo usuário, a imagem não é gerada novamente, mas sim carregada do *cache* de imagens, o que torna a transição entre os diferentes estados de *zoom* mais rápida.

Figura 28 – Um grafo de memória no Reflector.



Fonte: Autoria própria.

3.8 SÍNTESE

Neste capítulo, foram apresentados os conceitos e técnicas utilizados pela abordagem Reflector e implementados na ferramenta homônima para criar visualizações que mostram o uso da memória de um sistema de software desenvolvido na linguagem Java.

Para a criação de tais visualizações, foram utilizadas a linguagem DOT e seu motor de desenho. Utilizando as funcionalidades de customização da linguagem, foram desenvolvidas ainda opções de filtragem e simplificação das representações gráficas geradas.

No próximo capítulo, o uso do Reflector em aplicações reais será demonstrado com a apresentação de dois estudos de caso feitos com o objetivo de explicar sua utilização.

4 ESTUDO DE CASO

Neste capítulo serão apresentados dois estudos de caso feitos para esclarecer o funcionamento da ferramenta Reflector. Para demonstrar e avaliar nossa abordagem, aplicamos a ferramenta a duas aplicações: um teclado virtual para usuários com deficiências motoras e um classificador de dados da popular ferramenta de mineração de dados WEKA (HALL *et al.*, 2009). O código-fonte utilizado no estudo de caso que utiliza a ferramenta WEKA está disponível sem custo no site dos desenvolvedores.

4.1 TECLADO VIRTUAL

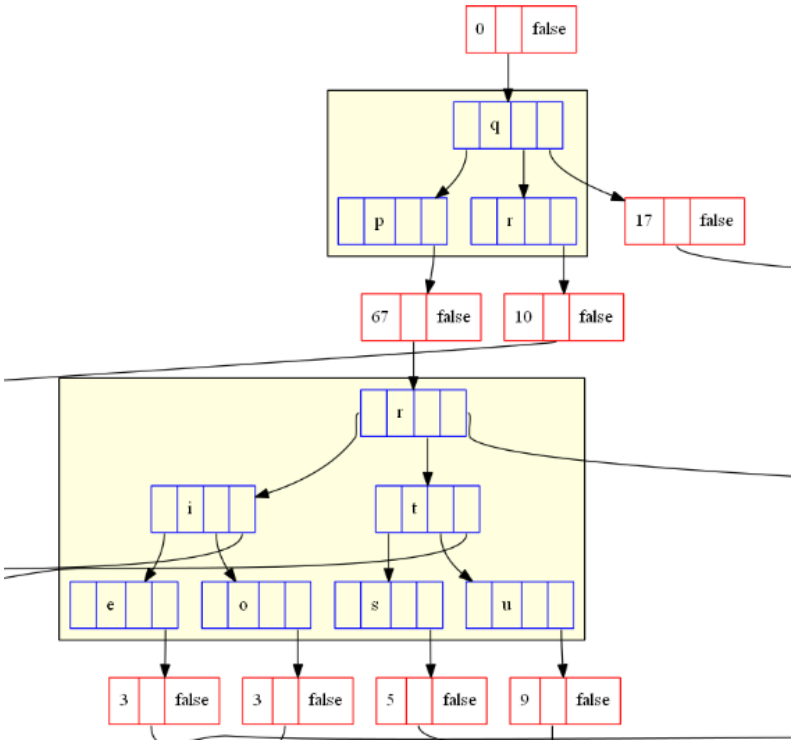
Uma das aplicações escolhidas para o estudo de caso é um teclado virtual para usuários com deficiências motoras. Ela foi escolhida para destacar algumas funcionalidades da ferramenta na tarefa de tornar mais fácil a visualização do grafo de memória de uma estrutura de dados relativamente complexa. A aplicação contém um dicionário de cem palavras armazenado em uma árvore digital (BRASS, 2008), que utiliza uma árvore binária balanceada de busca como uma estrutura secundária de armazenamento. Acreditamos que as representações visuais geradas com a aplicação da ferramenta Reflector à estrutura demonstra várias das funcionalidades mais interessantes da ferramenta de forma bastante interessante. A Figura 29 mostra uma imagem parcial do grafo de memória da árvore digital.

Note que, no grafo mostrado na Figura 29, foram utilizados filtros para destacar as duas classes usadas na aplicação, atribuindo uma cor diferente para cada uma delas, de forma que os objetos poder ser distintos claramente. Foi utilizada também a simplificação automática que agrupa objetos da mesma classe se eles são conectados uns aos outros. Esses grupos foram destacados em uma cor diferente para mostrar claramente seu papel nessa implementação particular da estrutura de árvore digital. Nesse caso, os grupos de objetos destacados em retângulos amarelos são árvores binárias balanceadas de busca que armazenam caracteres de um alfabeto e as referências associadas a esses caracteres, para economizar espaço na memória e acelerar a busca por verbetes no dicionário.

A Figura 30 mostra trechos do código-fonte usado na aplicação. Os vértices destacados em azul na Figura 29 são objetos da classe *BinTree*, cujos campos são mostrados

na Figura 30 (a) e os vértices destacados em vermelho são objetos da classe *Trie*, cujos campos podem ser vistos na Figura 30 (b).

Figura 29 – Árvore digital com filtros de destaque e agrupamento.



Fonte: Autoria própria.

Figura 30 – Trechos de código-fonte da aplicação.

```
public class BinTree {
    private BinTree left;
    private char letter;
    private BinTree right;
    private Trie next;
```

(a)

```
public class Trie {
    private int count;
    private BinTree children;
    private boolean end;
```

(b)

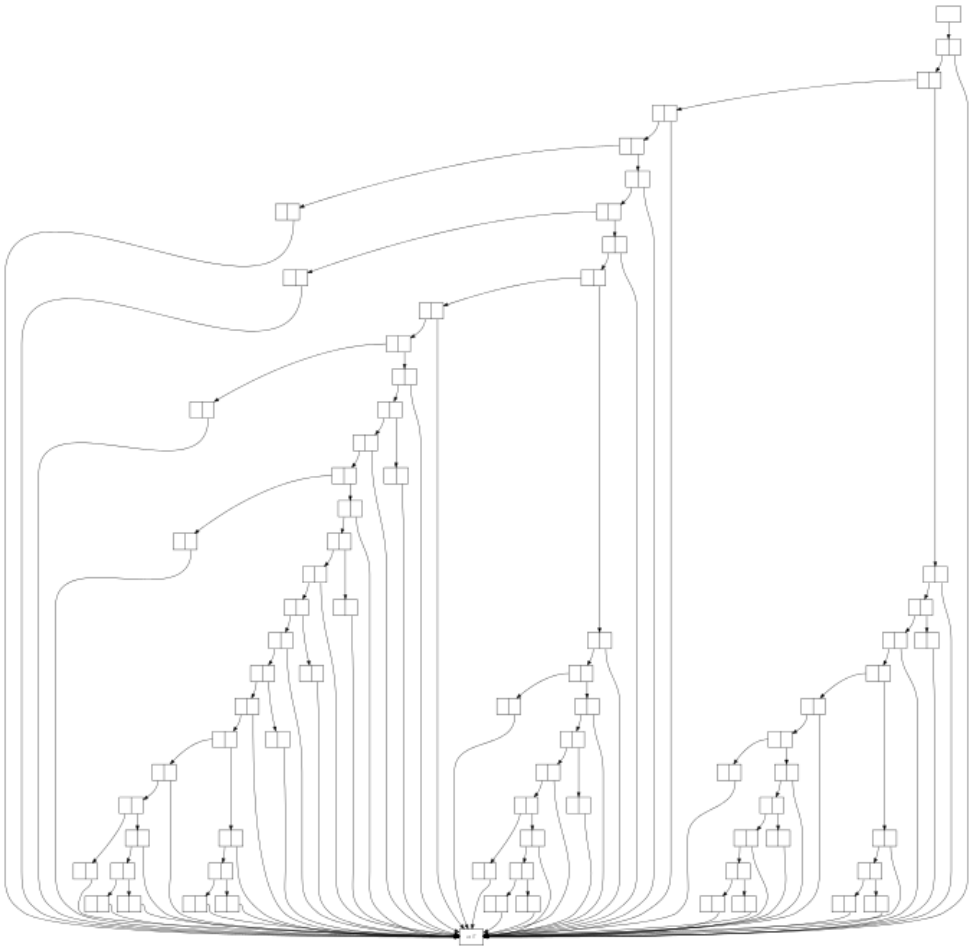
Fonte: Autoria própria.

Ainda na Figura 29, é importante notar que o número de partições na representação visual de cada objeto é igual ao seu número de campos (ou ao número de campos não filtrados pelo usuário). Eles aparecem na imagem na mesma ordem em que são declarados no código-fonte, a não ser que o usuário utilize o filtro de atributos mencionado anteriormente. Nesse caso, os campos são representados visualmente na mesma ordem em que foram declarados no filtro.

4.2 CLASSIFICADOR WEKA

WEKA é uma coleção de algoritmos de aprendizado de máquina para tarefas de mineração de dados (HALL *et al.*, 2009). É um software de código aberto. Para este estudo de caso, um dos muitos classificadores do WEKA foi analisado. Para isso, foi necessário incluir uma chamada ao Reflector no código-fonte da classe *ClassifierPanel*, mais precisamente no método *startClassifier*. A variável *classifier* foi escolhida para ser observada pelo Reflector, logo após uma chamada ao método *buildClassifier*. Isso inicia a interface gráfica do Reflector, que é executada no *thread* principal do WEKA e, portanto, interrompe sua execução até que a janela do Reflector seja fechada.

Figura 31 – Classificador REPTree do WEKA.



Fonte: Autoria própria.

A Figura 31 mostra a representação gráfica gerada. Para simplificar a imagem resultante, foram aplicados filtros a duas classes às quais o classificador utilizado faz referências. A classe *weka.classifiers.trees.REPTree.Tree* teve todos os seus atributos removidos pelo filtro de seleção, exceto pelos atributos *m_Successors* e *m_Info*, e a classe *weka.core.Instances* teve todos os seus atributos removidos pelo mesmo filtro, exceto pelo atributo *FILE_EXTENSION*.

4.3 RESULTADOS

O estudo de caso do teclado virtual demonstrou que as técnicas de visualização da abordagem proposta podem facilitar a identificação das estruturas de dados utilizadas na implementação de sistemas. Essa característica é de grande importância para a compreensão da implementação de um software e, consequentemente, de grande valor para desenvolvedores na tarefa de manter ou evoluir um sistema de software.

Já o estudo de caso do classificador da ferramenta WEKA demonstrou o poder de simplificação fornecido pelo Reflector através do filtro de seleção. Com a utilização do filtro, foi possível a geração de uma imagem pouco carregada visualmente. Uma representação gráfica utilizando todos os campos das classes do classificador teria um tamanho impraticável, alta carga visual, e apresentaria, inevitavelmente, muitas informações irrelevantes para o usuário, como as diversas constantes utilizadas pelas classes para cálculos pertinentes ao funcionamento do classificador.

5 CONCLUSÃO

Neste capítulo serão apontadas as contribuições deste trabalho, discutidos os benefícios da abordagem proposta, e apontadas as direções para trabalhos futuros.

5.1 CONTRIBUIÇÕES

A visualização de software é uma ferramenta poderosa em processos que exigem uma compreensão profunda da forma como um sistema de software funciona. Ao representar o software de forma visual, a visualização permite que o desenvolvedor perceba melhor sua lógica e sua estrutura. Tais representações visuais são potencialmente mais informativas para os usuários do que representações descritivas, pois permitem explorar suas habilidades.

Este trabalho apresentou uma abordagem para a geração automática de visualizações da memória utilizada por aplicações Java, que é implementada em uma ferramenta de visualização de software chamada Reflector. Ela usa a API Java Reflection para analisar a memória da Máquina Virtual Java. A partir de tal análise, a ferramenta gera uma representação visual da memória usada por qualquer aplicação Java, usando seus objetos e os relacionamentos entre eles para construir uma estrutura de grafo e criar uma representação gráfica de acordo com especificações feitas pelo usuário. A ferramenta tem uma interface de usuário através da qual os usuários podem configurar filtros e simplificações que podem produzir imagens mais interessantes e informativas para um propósito específico. Ela permite ainda que o usuário salve a imagem para visualização posterior e que ele salve uma especificação do grafo na linguagem DOT, que pode ser utilizada em muitas outras ferramentas de visualização.

As visualizações geradas procuram resolver um problema comum: a abstração dos dados de baixo nível. A grande quantidade e complexidade de dados de execução obtidos de sistemas de software de grande porte podem dificultar bastante a criação de visualizações informativas para os usuários. Com a abordagem Reflector, é possível criar visualizações concretas de dados abstratos e filtrar as visualizações de forma a diminuir a quantidade de informação e dar maior importância a dados que são de maior relevância para um determinado propósito.

5.2 TRABALHOS FUTUROS

A lista a seguir mostra problemas percebidos durante o desenvolvimento do trabalho.

- Análise dinâmica: uma possível extensão do Reflector é a inclusão de uma forma de visualização que suporte um maior nível de interação do usuário com as representações gráficas geradas. Por exemplo, uma solução que permita que o usuário reorganize manualmente a posição dos vértices do grafo na representação gráfica final utilizando o mouse.
- Extração de padrões: padrões topológicos podem ser encontrados nos mais diversos tipos de software orientados a objeto, e compõem uma área de pesquisa. Um possível trabalho futuro é a inclusão de simplificações que identifiquem padrões e os represente de forma distinta nas visualizações geradas.
- Visualização: a quantidade de dados e os tipos de dados utilizados em aplicações Java de grande porte exigem melhoramentos nas técnicas de visualização. Em particular, trabalhos futuros devem ser feitos na apresentação e navegação das representações gráficas.

REFERÊNCIAS

ACKOFF, R. L. From Data to Wisdom. **Journal of Applied Systems Analysis**, Lancaster, v. 16, p. 3—9, 1989.

BALL, T.; EICK, S. Software Visualization in the Large. **IEEE Computer**, Estados Unidos, v. 24, n. 4, p. 33—43, 1996.

BOEHM, B.; BECK, K. The Changing Nature of Software Evolution. **IEEE Software**, Estados Unidos, vol. 27, n. 4, pp. 26—28, 2010.

BRASS, P. *Advanced Data Structures*. Cambridge: Cambrige University Press, 2008.

BRON, C.; KERBOSCH, J. Algorithm 457: Finding All Cliques of an Undirected Graph. **Communications of the ACM**, Estados Unidos, v. 16, n. 9, p. 575—577, 1973.

CAMERON, G.; JU-PAK, K.; KIM, B. Information Pollution. **Newspaper Research Journal**, Estados Unidos, v. 21, n. 1, p. 65—76, 2000.

CHEN, M.; EBERT, D.; HAGEN, H.; LARAMEE, R.; VAN LIERE, R.; MA, K.; RIBARSKY, W.; SCHEUERMANN, G.; SILVER, D. Data, information, and knowledge in visualization. **IEEE Computer Graphics and Applications**, Estados Unidos, v. 29, n. 1, p. 12—19, 2009.

CARD, S.; MACKINGLAY, J.; SHNEIDERMAN, B. *Readings in Information Visualization: Using Vision to Think*. São Francisco: Morgan Kaufmann, 1999.

CRAFT, B; CAIRNS, P. Directions for Methodological Research in Information Visualization. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALIZATION, 12., 2008, Londres. **Proceedings...**, Estados Unidos: IEEE Computer Society, 2008. p. 44–50.

DUCASSE, S.; LANZA, M. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. **IEEE Transactions on Software Engineering**, Estados Unidos, v. 29, n. 9, p. 782—795, 2003.

EDEN, A. H. LePUS: A Visual Formalism for Object-Oriented Architectures. In: WORLD CONFERENCE ON INTEGRATED DESIGN AND PROCESS TECHNOLOGY, 6., 2002, Pasadena, California. **Proceedings...**, Pasadena, California: Society for Design and Process Science, 2002. pp. 26–30.

EINSFELD, K.; EBERT, A.; WOLLE, J. Modified Virtual Reality for Intuitive Semantic Information Visualization. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 12., 2008, Londres, Inglaterra. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2008. p. 515–520.

FEIJS, L.; DE JONG, R. 3D Visualization of Software Architectures. **Communications of the ACM**, Estados Unidos, v. 41, n. 12, p. 73—78, 1998.

FILONIK, D.; BAUR, D. Measuring Aesthetics for Information Visualization. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 13., 2009, Barcelona, Espanha. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2009, p. 579–584.

GALLAGHER, K.; HATCH, A.; MUNRO, M. Software Architecture Visualization: An Evaluation Framework and Its Application. **IEEE Transactions on Software Engineering**, Estados Unidos, v. 34, n. 2, p. 260–270, 2008.

GANSNER, E. R.; KOUTSOFIOS, E.; NORTH, S. C.; VO, K. P. A Technique for Drawing Directed Graphs. **IEEE Transactions on Software Engineering**, Estados Unidos, v. 19, n. 3, p. 214–230, 1993.

GARG, S.; VAN MOORSEL, A.; VAIDNYANATHAN, K.; TRIVEDI, K. S.; A Methodology for Detection and Estimation of Software Aging. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINERRING, 9., 1998, Paderborn, Alemanha. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 1998, p. 283—292.

GLEICHER, M.; ALBERS, D.; WALKER, R.; JUSUFI, I.; HANSEN, C. D.; ROBERTS, J. C. Visual Comparison for Information Visualization. **Information Visualization**, Filadélfia, Estados Unidos, v. 10, n. 4, p. 289–309, 2011.

GRUNDY, J.; HOSKING, J. High-level Static and Dynamic Visualisation of Software Architectures. In: INTERNATIONAL SYMPOSIUM ON VISUAL LANGUAGES, 16., 2000, Seattle, Estados Unidos. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2000, p. 5–12.

HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WITTEN, I.H. The WEKA Data Mining Software: An Update. **SIGKDD Explorations Newsletter**, Nova Iorque, Estados Unidos, vol. 11, n. 1, pp. 10–18, 2009.

HEER, J.; CARD, S.; LANDAY, J. Prefuse: A Toolkit for Interactive Information Visualization. In: SIGCHI CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS, 23., 2005, Portland, United States. **Proceedings...**, Nova Iorque, Estados Unidos: ACM, 2005, p. 421–430.

HEER, J.; SHNEIDERMAN, B. Interactive Dynamics for Visual Analysis. **Queue**, Estados Unidos, vol. 10, n. 2, p. 30, 2012.

HUANG, J.; LI, Y.; ZHANG, J.; YU, J. Developing Novel Design Patterns in Information Visualization for Mobile Health Systems. In: INTERNATIONAL CONFERENCE ON BIOMEDICAL ENGINEERING AND INFORMATICS, 3., 2010, Yantai, China. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2010, p. 2748–2752.

HUTCHINS, E.; LINTERN, G. *Cognition in the Wild*, Cambridge: MIT Press Cambridge, 1995.

KEIM, D.; KOHLHAMMER, J.; ELLIS, G.; MANSMANN, F. *Mastering the Information Age - Solving Problems with Visual Analytics.* Goslar, Alemanha: Eurographics Association, 2010.

KELLEY, S.; AFTANDILIAN, E.; GRAMAZIO, C.; RICCI, N.; SU, D. L.; GUYER, S. Z. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. **Information Visualization**, Filadélfia, Estados Unidos, 2012.

LAU, A.; MOERE, A. Towards a Model of Information Aesthetics in Information Visualization. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 11., 2007, Zurique, Suíça. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2007, p. 87–92.

LEHMAN, M. M.; RAMIL, J. F.; WERNICK, P. D., PERRY, D. E., TURSKI, W. M., Metrics and Laws of Software Evolution – The Nineties View. In: INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, 11. 1997, Albuquerque, Estados Unidos. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2007, p. 20—32.

LIU, Z.; NERSESSIAN, N.; STASKO, J. Distributed Cognition as a Theoretical Framework for Information Visualization. **IEEE Transactions on Visualization and Computer Graphics**, Estados Unidos, v. 14, n. 6, p. 1173–1180, 2008.

LIVSHITS, B.; WHALEY, J.; LAM, M. Reflection Analysis for Java. **Programming Languages and Systems**, Estados Unidos, p. 139—160, 2005.

MALETIC, J.; MARCUS, A. CFB: A Call For Benchmarks – for Software Visualization. In: IEEE WORKSHOP OF VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS, 2., 2003, Amsterdã, Holanda. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2003, p. 108—113.

MAZZA, R. *Introduction to Information Visualization*. Nova Iorque, Estados Unidos: Springer, 2009.

MCCORMICK, B.H.; DEFANTI, T.; BROWN, M.D. Visualization in Scientific Computing. **Computer Graphics**, v. 21, p. 1–14, 1987.

MOERE, A. Beyond The Tyranny of The Pixel: Exploring The Physicality of Information Visualization. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 12., 2008, Londres, Inglaterra. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2008, p. 469–474.

MYERS, C.; DUKE, D. A Map of the Heap: Revealing Design Abstractions in Runtime Structures. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE VISUALIZATION, 5., 2010, Salt Lake City, Estados Unidos. **Proceedings...**, Nova Iorque, Estados Unidos: ACM, 2010, p. 63—72.

NING, Z.; WENXING, H.; SITING, Z.; LI, T. A Solution for an Application of Information Visualization in Telemedicine. In: INTERNATIONAL CONFERENCE ON COMPUTER

SCIENCE & EDUCATION, 7., 2012, Melbourne, Australia. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2012, p. 407–411.

PARNAS, D. L.; Software Aging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 16., 1994, Sorrento, Itália. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 1994, p. 279—287.

PERRY, D. E.; WOLF, A. L. Foundations for the Study of Software Architecture**. ACM SIGSOFT Software Engineering Notes**, Nova Iorque, Estados Unidos, vol. 17, n. 4, pp. 40—52, 1992.

PINZGER, M. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. 2005. 149 f. Tese (Doutorado em Ciências Técnicas) – Vienna University of Technology, Viena, Áustria.

PINTO, M.; RAPOSO, R.; RAMOS, F. Comparison of Emerging Information Visualization Tools for Higher Education. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 16., 2012, Monpellier, França. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2012, p. 100–105.

RABELO, E.; DIAS, M.; FRANCO, C.; PACHECO, R. Information Visualization: Which Is The Most Appropriate Technique to Represent Data Mining Results? In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE FOR MODELLING CONTROL & AUTOMATION, 2008, Viena, Áustria. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2008, p. 1228–1233.

REISS, S. P. Visualizing the Java Heap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 32., 2010, Cape Town, África do Sul. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2010, p. 251—254.

SAWANT, A. P.; BALI, N. SoftArchViz: A Software Architecture Visualization Tool. In: INTERNATIONAL WORKSHOP ON VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS, 4., Banff, Canada. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 2007, p. 154—155.

SHARIR, M. A Strong-Connectivity Algorithm and its Applications in Data Flow Analysis. **Computers & Mathematics with Applications**, Amsterdã, Holanda, vol. 7, pp. 67–72, 1981.

SIM, S. E.; CLARKE, C. L. A.; HOLT, R.C.; COX, A.M. Browsing and Searching Software Architectures. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 15., Oxford, Inglaterra. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, 1999, p. 381—390.

SOMMERVILLE, I. *Software Engineering*. Nova Jérsei: Pearson Education, 2011.

SPRITZER, A.; DAL SASSO FREITAS, C. Design and Evaluation of MagnetViz -- A Graph Visualization Tool. **IEEE Transactions on Visualization and Computer Graphics**, Estados Unidos, vol. 18, pp. 822–835, 2012.

STEINDL, C. *Program Slicing for Object-Oriented Programming Languages*. Nova Jérsei: Citeseer, 2000.

TEYSEYRE, A.; CAMPO, M. An Overview of 3D Software Visualization. **Transactions on Visualization and Computer Graphics**, Estados Unidos, vol. 15, pp. 87–105, 2009.

TOBIASZ, M.; ISENBERG, P.; CARPENDALE, S. Lark: Coordinating Co-located Collaboration with Information Visualization. **Transactions on Visualization and Computer Graphics**, Estados Unidos, vol. 16, pp. 1065—1072, 2009.

WARE, C. *Information Visualization: Perception for Design*. Amsterdã, Holanda: Morgan Kaufmann, 2012.

WEISER, M. Program Slicing. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 5., 1981, San Diego, Estados Unidos. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, p. 439–449, 1981.

WOHLFART, E.; AIGNER, W.; BERTONE, A.; MIKSCH, S. Comparing Information Visualization Tools Focusing on the Temporal Dimensions. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALISATION, 12., 2008, Londres, Inglaterra. **Proceedings...**, Washington, Estados Unidos: IEEE Computer Society, p. 69–74, 2008.

ZHANG, J. *Visualization for Information Retrieval*. Nova Iorque, Estados Unidos: Springer, 2008.