UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"

Pós-Graduação em Ciência da Computação

Fernanda Madeiral Delfim

Uma Abordagem Usando Visualização de Software como Apoio à Refatoração para Aspectos

UNESP
2013

Fernanda Madeiral Delfim

Uma Abordagem Usando Visualização de Software como Apoio à Refatoração para Aspectos

Orientador: Prof. Dr. Rogério Eduardo Garcia

Dissertação apresentada ao Instituto de Biociências, Letras e Ciências Exatas (IBILCE – UNESP – São José do Rio Preto) como parte dos requisitos para a obtenção do título de mestre em Ciência da Computação.

UNESP
2013

Fernanda Madeiral Delfim

Uma Abordagem Usando Visualização de Software como Apoio à Refatoração para Aspectos

Dissertação apresentada ao Instituto de Biociências, Letras e Ciências Exatas (IBILCE – UNESP – São José do Rio Preto) como parte dos requisitos para a obtenção do título de mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Rogério Eduardo Garcia
UNESP – Presidente Prudente
Orientador

Prof. Dr. Fabiano Cutigi Ferrari
Universidade Federal de São Carlos

Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Presidente Prudente, 06 de Agosto de 2013

Dedico este trabalho
à Marta

Agradecimentos

A Deus, por ter me guiado sempre, principalmente nos momentos mais difíceis.

Ao meu orientador, Rogério Eduardo Garcia, que me "pegou para criar" sem ao menos me conhecer. Obrigada pelos ensinamentos, pela paciência e pelo meu direcionamento. Você contribuiu muito para a minha formação (e espero que o verbo "contribuir" no passado se torne futuro). Desculpe-me pelo trabalho, mas filho dá trabalho mesmo (risos).

Aos meus pais, Marta e Gualdemir, por terem me incentivado aos estudos desde pequena. Obrigada pelo apoio para a realização deste trabalho, pela compreensão nos meus dias de mau humor e pelo apoio financeiro incondicional.

Ao Márcio, não somente por ter feito ícones para a ferramenta e imagens vetorizadas para o texto (risos), mas por ter estado presente em todos os momentos durante o mestrado – agradeço especialmente pelos momentos mais difíceis e desesperadores, pois sei que não é nada fácil lidar com o meu gênio. Obrigada por todo apoio, por toda compreensão, por todos os conselhos, por todos os abraços.

A todos os amigos do mestrado, Li, Jorge, Álvaro, Neto, Helton e Vanessa, por terem compartilhado comigo momentos incríveis. As viagens para os WPPGCC – em Bauru (2011), Rio Preto (2012) e Rio Claro (2013) – foram inesquecíveis.

À Li, que me acolheu quando eu entrei no mestrado, pois não sabia nem andar pelo campus da Unesp (risos), e que também foi minha companheira de viagem durante um ano para a realização das disciplinas em Bauru. Eu nunca irei esquecer da nossa correria para ir para Bauru e voltar para Prudente, dos cafés da manhã agradáveis com você no hotel e dos cafés da tarde na cantina da Unesp de Bauru. Obrigada por todo apoio e por ter me aguentado nos momentos mais difíceis.

Ao Jorge, que chegou um ano depois da minha entrada no mestrado, e desde então a família passou a ser completa. Obrigada por todo apoio e por ter me aguentado nos momentos mais difíceis.

A todos os amigos que me incentivaram, apoiaram e comemoraram comigo as etapas parciais con-

cluídas, em especial à Gi, minha eterna amiga, à Flávia, à Bruna e ao Cassara, que conheci durante o período de mestrado e que trouxeram mais felicidades para a minha vida.

A todos os colegas do mestrado em Matemática pelos momentos na sala 13, em especial ao Clóvis e ao Merejolli, que são duas peças das quais nunca me esquecerei.

Aos meus professores da graduação que me apoiaram em relação ao mestrado, em especial ao Francisco Assis da Silva e à Maria José Liberati pelos conselhos, e à Ana Paula e ao Ivam Resina pela carta de recomendação para o processo seletivo.

Aos professores que compuseram as bancas de avaliação do meu trabalho – Celso Olivete Junior e Danilo Medeiros Eler pela participação na banca de Estudos Especiais; Ronaldo Celso Messias Correia pela participação na banca de qualificação; e Fabiano Cutigi Ferrari por ter aceito o convite para participar da banca da defesa. Em especial, ao Marcelo de Almeida Maia, pela participação na banca de qualificação e por ter aceito o convite para participar da banca da defesa.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), pelo apoio financeiro.

"Qualquer tolo consegue escrever código que um computador consegue compreender. Bons programadores escrevem código que humanos conseguem compreender."
Martin Fowler

"O céu não está perto de suas mãos. Para chegar lá é preciso muito trabalho e, consequentemente, tempo e dedicação."
Rogério Eduardo Garcia

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

CCC – Interesse transversal (do inglês *Crosscutting Concern*)
CFG – Grafo de Fluxo de Controle (do inglês *Control Flow Graph*)
DUG – Grafo de Definição-Uso (do inglês *Definition-Use Graph*)
PDG – Grafo de Dependência de Programa (do inglês *Program Dependence Graph*)
SDG – Grafo de Dependência de Sistema (do inglês *System Dependence Graph*)
VA – Visualização de Algoritmo
VP – Visualização de Programa
VS – Visualização de Software

Resumo

A evolução de sistemas de software existentes para a tecnologia orientada a aspectos tem como primeiro passo a mineração de aspectos, que visa a identificar interesses transversais em código fonte, para serem encapsulados em aspectos. Diversas técnicas têm sido propostas para a mineração de aspectos, mas ainda com deficiências. Uma das causas dessas deficiências apontada na literatura é a apresentação inadequada dos resultados obtidos por tais técnicas. A Visualização de Software pode ser utilizada para analisar, interpretar e combinar resultados de técnicas de mineração de aspectos, sendo os resultados apresentados juntamente com características de programa. Neste trabalho é apresentada uma abordagem visual de múltiplas visões coordenadas com o propósito de prover um ambiente para a apresentação dos resultados gerados por técnicas de mineração de aspectos, para melhorar a compreensão do usuário ao analisá-los para futura refatoração para aspectos. As múltiplas visões coordenadas são utilizadas para permitir a análise: das associações baseadas em chamadas de métodos, em nível de classe e de método, permitindo a visualização da frequência de chamadas das unidades baseada na métrica *fan-in*; das dependências de controle e de dados entre instruções de programa; da estrutura de programa; de como conjuntos de instruções (fatias) são compostos em diversas classes; e do bytecode. O foco é investigar se a visualização contribui na compreensão de programas por meio dos resultados gerados usando as técnicas fatiamento de programa e análise de *fan-in*, propostas para minerar aspectos, de maneira complementar. Uma ferramenta de visualização de software, nomeada $SoftVis_{4CA}$ (*Software Visualization for Code Analysis*), foi desenvolvida para apoiar a abordagem visual proposta. O estudo preliminar mostrou que o modelo de coordenação proposto apoia a análise pela exploração de diferentes níveis de detalhes.

Abstract

The evolution of existing software systems to aspect-oriented technology has as first step the aspect mining, which aims to identify crosscutting concerns in source code to be encapsulated into aspects. Several techniques have been proposed for aspect mining, but still with shortcomings. One cause of these shortcomings pointed out in the literature is inadequate presentation of the results obtained by these techniques. Software Visualization can be used to analyze, interpret and combine results of aspect mining techniques, being the results presented with program characteristics. This work presents a visual approach of multiple coordinated views in order to provide an environment for the presentation of the results generated by aspect mining techniques, as well as to improve the understanding of the user to analyze them for future refactoring to aspects. The multiple coordinated views are used to allow the analysis: of associations based on method calls, at class and method levels, allowing visualization of the units call frequency based on *fan-in* metric; of the control and data dependencies between program instructions; of the program structure; of how instruction sets (slices) are composed in several classes; and of bytecode. The focus is to investigate whether visualization helps in program comprehension by the results generated using program slicing and *fan-in* analysis techniques, proposals for mining aspects in a complementary way. A software visualization tool, named $SoftVis_{4CA}$ (*Software Visualization for Code Analysis*), was developed to support the proposed visual approach. The preliminary study showed that the proposed coordination model supports the analysis by exploration of different levels of details.

CAPÍTULO 1

Introdução

A manutenção de software é relatada como uma atividade de custo alto (Mens e Tourwé, 2004), principalmente pelo esforço gasto para a compreensão do projeto e do código de um sistema, antes de sua modificação. Tal esforço normalmente ocorre quando o sistema possui um projeto ruim (Drozdz et al., 2006) ou o código é mal estruturado (Atkinson e King, 2005). Uma solução proposta para reduzir o esforço de manutenção é a *refatoração*[1], que é o processo de modificar a estrutura[2] do código de um sistema sem afetar o seu comportamento, mas melhorando sua estrutura interna (Fowler, 1999). Segundo Fowler (1999), a refatoração ajuda na melhoria do projeto e do código do software, removendo instruções desnecessárias e organizando as instruções na unidade modular ideal. Assim, é mais fácil *compreender* o software e permite que outras mudanças sejam feitas com mais facilidade (Opdyke, 1992) e, consequentemente, reduzir os custos a elas associados (Carneiro, 2003).

A refatoração requer a análise de código para a identificação de trechos de código potencialmente "ruins", isto é, as partes do código que podem ser modificadas para melhorar sua compreensão e sua facilidade de manutenção. Pesquisadores têm mapeado tais trechos de código em *bad smells*, que são indícios de características de software que podem tornar o software difícil de compreender, de evoluir e de manter (Fontana e Spinelli, 2011). A partir de um indício identificado, e confirmada a existência de um problema, o código pode ser modificado usando mecanismos pré-definidos de refatoração. Entretanto, alguns problemas – *código duplicado*, *alterações divergentes* e *cirurgia com rifle* (Fowler, 1999) (ou *solução espalhada* (Kerievsky, 2004)) – são resolvidos parcialmente em sistemas orientados a objetos, pois são problemas que indicam a presença da implementação de *interesses transversais* no código fonte, e o paradigma orientado a objetos não suporta adequadamente a modularização de tais interesses (Massicotte et al., 2007). Decorrente disso, foi proposto o *Desenvolvimento de Software Orientado a Aspectos*, com mecanismos e abstrações para modularizar interesses transversais

[1] O termo *refatoração* (verbo) é o processo de aplicar técnicas para melhorar a estrutura interna de um sistema existente sem alterar o seu comportamento, enquanto *refatoração* (substantivo) é referente a técnicas que melhoram aspectos não funcionais em um sistema existente (Neill e Gill, 2003).

[2] *Estrutura*, neste contexto, refere-se a maneira pela qual as partes de um sistema estão dispostos ou organizados.

por meio do encapsulamento da implementação desses interesses em uma unidade de modularização separada do código base, chamada *aspecto*. Desse modo, são apoiados a reutilização de código (Kiczales et al., 1997) e o aumento da coesão dos módulos (Clifton et al., 2007).

1.1 Formulação do Problema e Motivação

A evolução de sistemas de software existentes para a tecnologia orientada a aspectos ainda é um desafio. A primeira etapa para a introdução de aspectos em um sistema não-orientado a aspectos é identificar potenciais interesses transversais no sistema existente. A segunda etapa é reformular a implementação desses interesses em aspectos, no caso de um interesse transversal confirmado por um usuário. Tais etapas são atividades denominadas *mineração de aspectos* e *refatoração para aspectos*, respectivamente.

Este trabalho concentra-se na atividade de mineração de aspectos como um apoio a refatoração para aspectos. Técnicas como *análise de padrões recorrentes em rastros de execução* (Breu e Krinke, 2004) e em *grafo de fluxo de controle* (Krinke e Breu, 2005), *análise formal de conceito de rastros de execução* (Tonella e Ceccato, 2004) e de *identificadores* (Tourwé e Mens, 2004), *detecção de clones* (Roy et al., 2009), *análise de fan-in* (Marin et al., 2007), *análise dinâmica e regras de associação* (Abait et al., 2008), *análise de link* (Huang et al., 2010) e *fatiamento de programa* (Katti et al., 2012), têm sido propostas para minerar aspectos. Em geral, essas técnicas possuem deficiências similares, como: *precisão ruim* dos resultados, em que a porcentagem de candidatos a aspectos relevantes no conjunto total de candidatos relatados por uma dada técnica é relativamente baixa; *subjetividade* ao analisar os resultados, pois os resultados produzidos apresentam algumas ambiguidades, e dependendo da pessoa e da definição de aspecto que ela usa, pode ocorrer de uma pessoa dizer que algo é um candidato a aspecto enquanto que outra pessoa pode dizer que não é; *difícil comparabilidade* dos resultados de diferentes técnicas e; *difícil agregabilidade* entre elas. Uma das causas dessas deficiências é a *apresentação inadequada dos resultados* (Mens et al., 2008). Nesse contexto, abordagens e ferramentas para lidar com as deficiências das técnicas de mineração de aspectos atuais podem ser úteis.

1.2 Objetivo e Justificativa

A análise e a interpretação de resultados de técnicas de mineração de aspectos requer, também, a compreensão do programa, tendo em vista que tais resultados precisam ser confirmados e analisados por um usuário para posterior refatoração. A Visualização de Software é uma abordagem que pode ser utilizada para apresentar resultados de técnicas de mineração de aspectos, juntamente com características de software como, por exemplo, a organização do código fonte, as associações de diferentes níveis de abstração e como tais associações estão presentes no código fonte.

Segundo Stasko et al. (1998), entender um programa analisando sua forma textual pode ser uma tarefa difícil, dependendo do seu tamanho e da sua complexidade. Uma representação gráfica de um programa pode ajudar consideravelmente na compreensão do seu significado, metodologia e propósito, ajudando a entendê-lo. As características de software possuem níveis de abstração diferentes, assim como os resultados obtidos por técnicas de mineração de aspectos.

O objetivo deste trabalho é investigar se o uso de visualização de software melhora a apresentação dos resultados de técnicas de mineração de aspectos, no sentido de eficácia e eficiência na análise e na interpretação deles para que o usuário consiga usá-los como entrada para a refatoração para aspectos. Neste trabalho é apresentada uma abordagem visual utilizando técnicas de visualização de software e de interação para a apresentação de resultados de técnicas de mineração de aspectos, em um ambiente para exploração visual de características de programa.

1.3 Organização do Texto

Para expor os assuntos aqui tratados, esta dissertação encontra-se organizada como segue:

- No Capítulo 2 são apresentadas uma visão geral dos indícios no código fonte em que há a possibilidade de refatoração de código, a motivação para a introdução de aspectos em um sistema orientado a objetos e uma visão geral sobre mineração de aspectos, apresentando duas técnicas que são utilizadas no desenvolvimento deste trabalho. Adicionalmente, são apresentadas uma avaliação de técnicas de mineração de aspectos e uma comparação entre elas, bem como deficiências das técnicas e suas causas, que foram apontadas na literatura;

- No Capítulo 3 é introduzido o uso de visualização, bem como são apresentadas técnicas de visualização e de interação utilizadas neste trabalho. A motivação da utilização de múltiplas visões é introduzida, juntamente com técnicas de coordenação de múliplas visões. Por fim, é dada uma visão geral sobre visualização de software juntamente com a apresentação de ferramentas de visualização de software utilizadas para fins de Engenharia de Software;

- No Capítulo 4 é apresentada a ferramenta $SoftVis_{4CA}$ juntamente com o modelo de coordenação proposto, com foco em questões de implementação;

- No Capítulo 5 são apresentadas e explicadas as representações visuais obtidas por meio da implementação da ferramenta $SoftVis_{4CA}$. Além disso, são apresentados uma estratégia para a utilização das representações visuais e um estudo piloto mostrando a utilização delas por meio da estratégia;

- No Capítulo 6 são expostas as conclusões do trabalho.

CAPÍTULO 2

Refatoração para Aspectos

A *manutenção de software* implica na compreensão de sistema de software para a tomada de decisões de onde e de como realizar uma modificação no código, bem como para analisar o impacto de mudança e para não introduzir defeitos no sistema. No entanto, um projeto ruim ou com código mal estruturado prejudica a manutenção, pois são difíceis de *compreender* e de *modificar* (Drozdz et al., 2006, Atkinson e King, 2005). Indícios da existência de problemas em projeto e em código que contribuem para a dificuldade de manutenção têm sido definidos na literatura (descritos na Seção 2.1). A identificação de tais indícios em sistemas é importante, pois no caso da confirmação da existência de um problema, o código e/ou o projeto podem ser melhorados por meio do uso de *refatoração de código* e, assim, a manutenção de software pode ser facilitada.

Existem diversos mecanismos de refatoração de código que podem ser aplicados para eliminar problemas em um código orientado a objetos usando soluções do mesmo paradigma (Fowler, 1999, Kerievsky, 2004). Entretanto, alguns problemas são resolvidos apenas parcialmente usando orientação a objetos, pois esse paradigma não suporta adequadamente a modularização de *interesses transversais* (do inglês *Crosscutting Concerns – CCCs*). Uma solução alternativa é utilizar mecanismos e abstrações providos pelo *Desenvolvimento de Software Orientado a Aspectos* para encapsular a implementação de *CCCs* em módulos separados de código orientado a objetos, chamados *aspectos* (a motivação dessa tarefa é descrita na Seção 2.2). Como um auxílio, a mineração de aspectos pode ser utilizada para identificar potenciais *CCCs* em um código para posteriormente serem encapsulados em aspectos. Diversas técnicas têm sido propostas para a mineração de aspectos, e neste trabalho são apresentadas a *Análise de Fan-in* e o *Fatiamento de Programa* (subseções 2.3.1 e 2.3.2).

As técnicas propostas para minerar aspectos possuem naturezas diferentes devido às suas particularidades. Por meio de critérios definidos na literatura, foram realizadas a avaliação individual de cada técnica e a comparação entre elas (Seção 2.4). Deficiências e possíveis causas das deficiências de tais técnicas foram identificadas na literatura, e são descritas na Seção 2.5.

2.1 *Bad Smells*

Pesquisadores têm mapeado indícios no código e no projeto que potencialmente indicam um problema, chamados *bad smells* (Fowler, 1999), representando anomalias estruturais que geralmente tornam o programa mais difícil de entender e de alterar (Arcoverde et al., 2011). Os indícios foram agrupados com base em uma classificação[1] definada por Mäntylä e Lassenius (2006). Tal classificação possui os seguintes grupos e indícios:

- *Os inchadores*, que representam indícios de algo que avoluma o tamanho do código, como *método longo*, *classe grande*, *lista de parâmetros longa*, *obsessão por tipos primitivos*, *grupos de dados* (Fowler, 1999) e *complexidade condicional* (Kerievsky, 2004);

- *Os maus usadores de orientação a objeto*, que são os indícios de problemas que representam casos em que a solução não explora adequadamente os recursos de projeto orientado a objetos, como *instrução switch*, *herança recusada*, *classes alternativas com interfaces diferentes* e *atributo temporário* (Fowler, 1999);

- *Os impedidores de alterações*, que são os indícios que consideravelmente impedem a modificação ou o desenvolvimento de software. Isso significa que os indícios desse grupo violam a regra sugerida por Fowler (1999), que as classes e as possíveis modificações devem ter uma relação um-para-um, isto é, um tipo de modificação deve afetar apenas uma classe (Mäntylä, 2003, Mäntylä e Lassenius, 2006), como *alterações divergentes*, *cirurgia com rifle* (Fowler, 1999) (ou *solução espalhada* (Kerievsky, 2004)) e *hierarquias de herança paralela* (Fowler, 1999);

- *Os dispensáveis*, que representam algo desnecessário que pode, e deve, ser removido do código. Tais indícios podem ser divididos em dois tipos: classes dispensáveis, como *classes preguiçosas* e *classes de dados* (Fowler, 1999), e código dispensável, como *código duplicado*, *generalidade especulativa* (Fowler, 1999), *código morto* (Mäntylä e Lassenius, 2006), *explosão combinatória*, *solução excêntrica* (Kerievsky, 2004) e *parâmetro obsoleto* (Tourwé e Mens, 2003);

- *Os acopladores*, que são os indícios que apresentam alto acoplamento. Dentre eles, três indícios apresentam acoplamento elevado (Mäntylä, 2003, Mäntylä e Lassenius, 2006), como *inveja dos dados*, *intimidade inapropriada* e *cadeias de mensagem* (Fowler, 1999). Os *intermediários desnecessários* (Fowler, 1999), por outro lado, representam um problema que pode ser criado quando se tenta evitar alto acoplamento com delegações constantes;

- *Outros indícios*, que não se enquadram em nenhum outro grupo de indícios, como *biblioteca de classes incompleta*, *comentários* (Fowler, 1999), *exposição indecente* (Kerievsky, 2004) e *interface inapropriada* (Tourwé e Mens, 2003).

[1] Na classificação definada por Mäntylä e Lassenius (2006) foram agrupados 22 indícios de Fowler (1999) e 1 indício introduzido por Mäntylä et al. (2003). Kerievsky (2004) e Tourwé e Mens (2003) definiram outros indícios relevantes, que foram adicionados à classificação de Mäntylä e Lassenius (2006) neste trabalho.

Para cada indício identificado, o mesmo deve ser analisado para a confirmação da existência de problema no código e para a decisão sobre refatorá-lo. Em caso de refatoração, há mecanismos de refatoração pré-definidos em catálogos de refatorações que podem ser aplicados para eliminá-lo de um código orientado a objetos, usando soluções do mesmo paradigma (Fowler, 1999, Kerievsky, 2004). Entretanto, alguns problemas são resolvidos apenas parcialmente em sistemas orientados a objetos, como:

- *Cirurgia com rifle* (Fowler, 1999) (ou *solução espalhada* (Kerievsky, 2004)), em que ao fazer um tipo de alteração em um sistema, diversas classes devem ser alteradas;
- *Alterações divergentes* (Fowler, 1999), em que uma única classe precisa ser modificada por diversos tipos de modificações (ou por diferentes razões);
- *Código duplicado* (Fowler, 1999), em que há um mesmo trecho de código em mais de um método, uma classe ou um pacote, e caso houver a necessidade de uma mudança em tal trecho de código, é necessário mudá-lo em vários módulos diferentes.

O motivo desses indícios serem resolvidos parcialmente com a orientação a objetos é que eles são sintomas da implementação de *interesses transversais* (espalhamento e emaranhamento de código), e o paradigma orientado a objetos não suporta adequadamente a modularização de tais interesses (Massicotte et al., 2007).

2.2 Motivação para Refatoração para Aspectos

A *separação de interesses* (Dijkstra, 1976) é um princípio bem estabelecido em Engenharia de Software. A ideia é dividir o domínio de um sistema em partes com o intuito de lidar com cada parte isoladamente em todo o desenvolvimento de software usando abstrações diferentes fornecidas por linguagens, métodos e ferramentas. As abstrações básicas do Desenvolvimento de Software Orientado a Objetos são classes, objetos, métodos e atributos. No entanto, essas abstrações podem não ser suficientes para a separação de interesses transversais, que naturalmente entrecortam a modularidade de outros interesses. Sem meios adequados para a separação e a modularização, a implementação de interesses transversais tende a ser *espalhada* por diversos módulos do sistema e *emaranhada* com outros interesses. Assim, é reduzida a facilidade de compreensão, evolução e reutilização de artefatos de software (Sant'Anna et al., 2003).

O *Desenvolvimento de Software Orientado a Aspectos*[2] (Filman et al., 2005) foi proposto como uma abordagem para melhorar a separação de interesses no desenvolvimento de sistemas orientados a objetos, para apoiar a reutilização de código (Kiczales et al., 1997) e o aumento da coesão dos módulos (Clifton et al., 2007). Para tanto, o Desenvolvimento de Software Orientado a Aspectos fornece mecanismos e abstrações que tornam possível: 1) encapsular a implemenação dos *CCCs* em uma unidade de modularização, chamada *aspecto*, separada das unidades na qual são implementados os interesses centrais (por exemplo classes, no caso da orientação a objetos) e; 2) compor as unidades para produzir o sistema global.

[2]Os conceitos de Programação Orientada a Aspectos não são apresentados neste trabalho.

A evolução de sistemas de software existentes para a tecnologia orientada a aspectos (isto é, a introdução de aspectos para o encapsulamento da implementação de *CCCs*) ainda é um desafio. Primeiro é necessário identificar potenciais *CCCs* em um sistema existente e, então, reformular esses interesses em aspectos no sistema de software. Tal identificação e reformulação são do domínio de pesquisa da *mineração de aspectos* e da *refatoração para aspectos* (também conhecida como *refatoração de interesses tranversais* (Hannemann, 2006) e *aspectualização* (Mortensen, 2009)), respectivamente. Na Figura 2.1 é ilustrado o processo de introdução de aspectos em um sistema orientado a objetos que, a partir do código de um sistema orientado a objetos é realizada a mineração de aspectos, tendo como saída trechos de código candidatos a serem encapsulados em aspectos. Por meio da análise de tais candidatos, um usuário decide sobre encapsular tais trechos de código em aspectos. No caso afirmativo, é realizada a refatoração para aspectos, gerando como saída do processo o sistema base contendo somente a implementação dos interesses primários e aspectos contendo a implementação de interesses transversais.

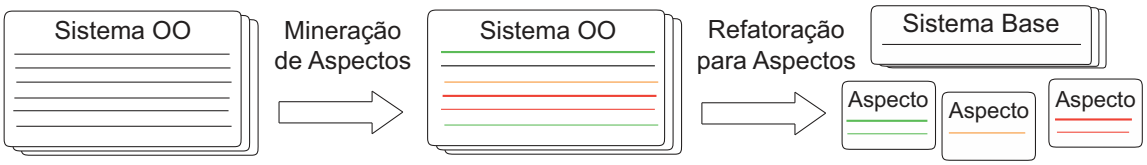


Figura 2.1: Processo de introdução de aspectos em um sistema orientado a objetos (Kellens et al., 2007).

2.3 Mineração de Aspectos

A mineração de aspectos busca elementos no código fonte de um sistema que pertencem à implementação de interesses que entrecortam estruturas de modularização de um sistema de software (*CCCs*), e que possivelmente podem ser melhorados pela utilização de soluções orientadas a aspectos. Tais elementos são chamados de *sementes*. Geralmente, a mineração de aspectos requer o envolvimento humano, permitindo considerar que as ferramentas de mineração de aspectos produzem *candidatos* a sementes, que podem ser transformados em *sementes confirmadas*, se aceitas por um humano, ou *não-sementes*, se rejeitadas. Uma não-semente é um falso positivo, e um falso negativo é uma parte de um *CCCs* conhecido, mas há perda decorrente das limitações inerentes de técnicas de mineração ou dos filtros específicos aplicados pelas mesmas. O desafio da mineração de aspectos é manter a porcentagem de sementes confirmadas no conjunto total de candidatos a sementes elevada, sem o aumento excessivo do número de falsos negativos (Marin et al., 2007).

Um processo genérico para implementação de técnicas de mineração de aspectos é ilustrado na Figura 2.2, em que é dividido em três etapas principais: *Pré-processamento*, que a partir de dados de um programa, como arquivos de código fonte e casos de teste, é responsável por obter os dados necessários e por gerar representações intermediárias para eles; *Mineração*, que é responsável pela execução de fato de alguma técnica, opcionalmente a partir da entrada de parâmetros requeridos por ela, para a geração de resultados; e *Pós-processamento*, em que filtros podem ser aplicados nos resultados, bem como pode ser realizada a organização deles para a apresentação ao usuário.

Diversas técnicas têm sido propostas para a mineração de aspectos – *análise de padrões recorrentes em rastros de execução* (Breu e Krinke, 2004) e em *grafo de fluxo de controle* (Krinke e Breu, 2005), *análise formal de conceito de rastros de execução* (Tonella e Ceccato, 2004) e de *identificadores* (Tourwé e Mens, 2004), *detecção de clones* (Roy et al., 2009), *análise de fan-in* (Marin et al., 2007), *análise dinâmica e regras de associação* (Abait et al., 2008), *análise de link* (Huang et al., 2010) e *fatiamento de programa* (Katti et al., 2012). Neste trabalho são apresentadas as técnicas *Análise de Fan-in* e *Fatiamento de Programa* nas duas próximas subseções.

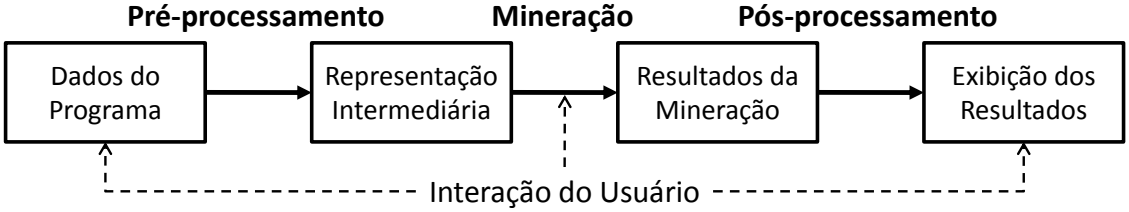


Figura 2.2: Processo genérico das ferramentas de mineração de aspectos.

2.3.1 Análise de *Fan-in*

Marin et al. (2007) propuseram o uso da métrica *fan-in* para tentar capturar código espalhado em nível de método. A suposição é de que *CCCs* são implementados em métodos exclusivos para eles e, então, esses métodos são invocados por vários módulos, dando-lhes um alto valor *fan-in* – o valor *fan-in* de um método é o número de métodos distintos que podem invocar o método sob análise, sendo que as informações necessárias para esse cálculo são adquiridas estaticamente do código fonte (Marin et al., 2007). A análise de *fan-in* proposta consiste em três etapas:

1. Computação automática da métrica *fan-in* para todos os métodos;

2. Filtragem do conjunto de métodos resultantes: (i) restringir o conjunto de métodos para aqueles que possuem um valor *fan-in* acima de um certo limiar; (ii) filtrar métodos de obtenção e modificação; (iii) filtrar métodos utilitários, métodos de manipulação de coleções, etc;

3. Análise dos métodos restantes para determinar quais deles fazem parte da implementação de um interesse transversal. Exemplos de invocações de um método com um alto valor *fan-in* a partir de locais de invocação que podem ser capturados por construções em aspectos, incluem: (i) as invocações que sempre ocorrem no início ou no final de um método; (ii) as invocações que ocorrem em métodos que são todos refinamentos de um método abstrato único como, por exemplo, de contratos exercidos por meio de hierarquias de classe; (iii) as invocações que ocorrem nos métodos com nomes semelhantes, como tratadores de eventos, e; (iv) todas as invocações que ocorrem nos métodos de implementação de um determinado papel, como em objetos-ouvinte que se registram como observadores de um estado objeto-sujeito.

Em uma reimplementação orientada a aspectos, o método constitui parte do comportamento a ser injetado por um aspecto (*advice*), e o local da invocação corresponde ao contexto do programa que precisa ser capturado (usando *pointcut*).

2.3.2 Fatiamento de Programa

Uma *fatia* de programa é um subconjunto de instruções do programa relativo a um subconjunto de variáveis de programa, isto é, é realizada a separação de um subconjunto de comportamento do programa. O processo de *fatiamento* de programa consiste em identificar as instruções que formam a *fatia* (identificação da *fatia*) e em isolar essas instruções em um programa independente (extração da *fatia*). Assim, o *fatiamento* de programa é semelhante ao processo de mineração de aspectos e refatoração (Katti et al., 2012).

As propriedades de *fatiamento de programa* são:

- *Estático e dinâmico* – No *fatiamento* estático é realizada a análise estática do código fonte para calcular a *fatia*, e no *fatiamento* dinâmico é realizada a execução do código fonte para uma determinada entrada e é calculada a *fatia* que é válida apenas para tal execução;

- *Forward e backward* – *Backward* é a maneira que foi introduzida originalmente por Weiser (1984), em que uma *fatia* contém todas as instruções que podem ter influenciado uma determinada variável em um determinado ponto de interesse. Por outro lado, *forward* contém todas as instruções que podem ser influenciadas por uma determinada variável;

- *Intra-procedimento e inter-procedimento* – As *fatias* podem se estender por um procedimento único (*intra-procedimento*) ou múltiplos procedimentos (*inter-procedimento*) de um programa. O *fatiamento* intra-procedimento calcula *fatias* dentro de um procedimento único. Por outro lado, o *fatiamento* inter-procedimento pode calcular *fatias* que abrangem procedimentos, podendo ser de diferentes classes e pacotes, quando se trata de *fatiamento* de programas orientados a objetos.

Katti et al. (2012) descrevem um algoritmo *crude*[3] para mineração e extração de aspectos usando *fatiamento de programa*, que envolve as seguintes etapas:

1. Identificação de todos os interesses no sistema usando *fatiamento de programa*;

2. A execução de alguma técnica de análise de programa como *Java Debugger Architecture*, *Aspects*, *Byte Code Instrumentation*, *Source Code Instrumentation*, etc. O mecanismo de reflexão da ferramenta de análise permite o acesso ao local exato onde um interesse é transversal com os outros. Assim, podem ser descobertos *CCCs*;

3. Se for utilizada a análise dinâmica para *fatiamento*, as ferramentas de análise devem ser reforçadas para incluir capacidades de reflexão e *pointcuts* amplos. Por exemplo, se *AspectJ* for usado, há a necessidade de obter variável local, definir *pointcuts*, e usar reflexão para identificar instruções em vez de números de linha;

4. Definição e execução de casos de teste para cada interesse, de modo que a análise de dados requerida para o *fatiamento* aconteça (tal tarefa pode ser automatizada usando alguma ferramenta de geração/execução de casos de teste);

[3] *Crude*, nesse contexto, significa que o algoritmo não é detalhado, isto é, foi descrito de uma maneira geral.

5. Detecção dos *CCCs* por meio da análise dos dados coletados durante o teste para determinar quais interesses são transversais;

6. Extração dos *CCCs* (*fatias*) em aspectos. A *fatia* obtida não pode ser inserida em um aspecto diretamente, pois ele deve ser dividido em *advices* diferentes. Se a *fatia* identificada é transversal, ela deve ser distribuída em diferentes *advices* e devem ser definidos os *join points* para a execução desses *advices*. Além disso, os *advices* e *pointcuts* para os *join points* devem ser encapsulados em um aspecto. A *fatia* obtida deve fazer sentido sintaticamente e deve ser introduzida corretamente no aspecto. Todos os dados relacionados com o interesse devem ser encapsulados em aspectos usando *inter type declarations*, se forem relacionados somente a esse interesse.

Apesar de Katti et al. (2012) terem descrito um algoritmo, ele ainda é muito superficial. Na primeira etapa, por exemplo, não foi definido qual critério é utilizado para a identificação de todos os interesses do sistema usando fatiamento de programa. Como os próprios autores apontam em seu trabalho, a pesquisa da utilização de fatiamento de programa para minerar aspectos e refatorá-los para aspectos ainda está em sua infância.

2.4 Avaliação e Comparação das Técnicas

Kellens et al. (2007) definiram um conjunto de *critérios* que permite a avaliação e a definição das particularidades de cada técnica proposta para mineração de aspectos, bem como a comparação entre elas, focando nas variabilidades delas. Com isso, é possível auxiliar os usuários das técnicas na escolha de uma técnica adequada para uma determinada situação, e ajudar os pesquisadores de mineração de aspectos a entender as diferenças entre elas. Os critérios definidos por Kellens et al. (2007) e que são utilizados neste trabalho, são:

- *Dados obtidos estaticamente versus dados obtidos dinamicamente.* Que tipo de dado a técnica analisa? Os dados de entrada podem ser obtidos estaticamente a partir da análise do código, os dados são obtidos dinamicamente pela execução do código, ou ambos?

- *Análise léxica, estrutural ou comportamental.* Que tipo de análise a técnica executa? Análise léxica, baseada em sequências de caracteres e expressões regulares (baseada em *token*), por exemplo, ou análise estrutural/comportamental, baseada em árvores de análise e envios de mensagens, por exemplo?

- *Granularidade.* Qual é o nível de granularidade dos candidatos a aspectos minerados? Algumas técnicas identificam aspectos em nível de classes e de métodos, outras identificam resultados mais refinados, considerando fragmentos de código como parte dos candidatos a aspectos.

- *Espalhamento e Emaranhamento.* Quais os sintomas de *CCCs* que a técnica de mineração de aspectos procura? Ela explicitamente procura por espalhamento, emaranhamento, ou ambos?

- *Envolvimento do usuário.* Que tipo de envolvimento do usuário é necessário para minerar aspectos? Qual o esforço que a técnica requer do usuário? O usuário tem que navegar por todos

os resultados a fim de indicar candidatos a aspectos viáveis? Existe entrada adicional exigida do usuário durante o processo de mineração de aspectos?

- *Pré-condições.* Quais condições (explícitas ou implícitas) devem ser satisfeitas pelos interesses no programa sob investigação para que uma técnica de mineração de aspectos particular consiga encontrar candidatos a aspectos adequados?

Com base nos critérios introduzidos por Kellens et al. (2007), as técnicas propostas para mineração de aspectos sumarizadas na Tabela 2.1 foram avaliadas individualmente, e por meio dos resultados individuais, as mesmas foram comparadas. Algumas técnicas consideradas para avaliação e comparação no trabalho de Kellens et al. (2007) foram desconsideradas neste trabalho, bem como foram adicionadas técnicas mais atuais. As técnicas avaliadas e comparadas por Kellens et al. (2007) estão sinalizadas por "($\star$)".

Na Tabela 2.1 são apresentados o tipo de dado de entrada (estático ou dinâmico) analisado e o tipo de análise realizada (baseada em *token*, estrutural ou comportamental) para cada técnica. A maioria das técnicas trabalham com dados estaticamente disponíveis. As técnicas *Análise de Padrões Recorrentes em Rastros de Execução*, *Análise Dinâmica e Regras de Associação*, *Análise Formal de Conceito de Rastros de Execução* e *Fatiamento de Programa* requerem dados obtidos pela execução do código em análise. Apenas *Fatiamento de Programa* trabalha com ambos os tipos de dado de entrada. Em relação ao tipo de análise, duas técnicas realizam a análise baseada em *token* da entrada de dados, em que elas dependem da suposição de que *CCCs* são frequentemente implementados pelo uso rigoroso de convenções de nomenclatura, e as outras nove técnicas analisam a entrada em nível estrutural ou comportamental.

Na Tabela 2.2 é apresentado o nível de granularidade das diferentes técnicas (classes, métodos ou fragmentos de código), e quais sintomas elas procuram (espalhamento e/ou emaranhamento). Com poucas exceções, a granularidade típica das técnicas é em nível de método e, portanto, a maioria das técnicas geram vários conjuntos de métodos, cada um representando uma potencial *semente* de aspecto. Apenas as técnicas de *Detecção de Clones* e *Fatiamento de Programa* detectam aspectos em nível de fragmentos de código e podem, portanto, fornecer resultados mais refinados sobre o código que pode ser encapsulado em *advices*. A única técnica que a granularidade é em nível de classes é a *Análise de Link*. Todas as técnicas usam o espalhamento como um indicador da presença de *CCC*, sendo apenas *Análise Formal de Conceito de Rastros de Execução* e *Fatiamento de Programa* que procuram por ambos os sintomas.

Na Tabela 2.3 é apresentado o tipo de envolvimento que é requerido do usuário por cada técnica, pois nenhuma é totalmente automática. Assim, todas as técnicas exigim que seus usuários naveguem pelos candidatos a aspectos resultantes a fim de encontrar aspectos. Algumas exigem que os usuários forneçam entrada apropriada como, por exemplo *Análise Dinâmica e Regras de Associação* e *Análise Formal de Conceito de Rastros de Execução*, que esperam como entrada um número de casos de uso.

Um critério importante para auxiliar na escolha de uma técnica adequada para minerar aspectos em um determinado sistema é determinar as suposições implícitas ou explícitas que a técnica faz sobre como *CCCs* são implementados. Na Tabela 2.4 são apresentadas tais suposições em termos de

pré-condições que um sistema deve satisfazer a fim de encontrar candidatos a aspectos viáveis com uma determinada técnica.

Tabela 2.1: Tipo de dado de entrada e tipo de análise de cada técnica.

	Tipo de dado de entrada		**Tipo de análise**		
	estático	dinâmico	*token*	estrut.	comport.
Análise de Padrões Recorrentes em Rastros de Execução ($\star$)	–	√	–	–	√
Análise de Padrões Recorrentes em Grafo de Fluxo de Controle	√	–	–	√	–
Análise Formal de Conceito de Rastros de Execução ($\star$)	–	√	–	–	√
Análise Formal de Conceito de Identificadores ($\star$)	√	–	√	–	–
Detecção de Clones (*Token*) ($\star$)	√	–	√	–	–
Detecção de Clones (Árvore Sintática Abstrata) ($\star$)	√	–	–	√	–
Detecção de Clones (Grafo de Dependência de Programa) ($\star$)	√	–	–	√	–
Análise de *Fan-in* ($\star$)	√	–	–	√	–
Análise Dinâmica e Regras de Associação	–	√	–	–	√
Análise de *Link*	√	–	–	√	–
Fatiamento de Programa	√	√	–	√	√

Tabela 2.2: Granularidade de e sintomas procurados por cada técnica.

	Granularidade			**Sintomas**	
	classe	método	fragmento	espalha.	emaranha.
Análise de Padrões Recorrentes (em Rastros de Execução e em Grafo de Fluxo de Controle)	–	√	–	√	–
Análise Formal de Conceito de Rastros de Execução	–	√	–	√	√
Análise Formal de Conceito de Identificadores	–	√	–	√	–
Detecção de Clones (*Token*, Árvore Sintática Abstrata e Grafo de Dependência de Programa)	–	–	√	√	–
Análise de *Fan-in*	–	√	–	√	–
Análise Dinâmica e Regras de Associação	–	√	–	√	–
Análise de *Link*	√	–	–	√	–
Fatiamento de Programa	–	–	√	√	√

Tabela 2.3: Envolvimento do usuário requerido por cada técnica.

Técnica	**Envolvimento do usuário**
Análise de Padrões Recorrentes (em Rastros de Execução e em Grafo de Fluxo de Controle)	Inspeção dos padrões recorrentes resultantes
Análise Formal de Conceito de Rastros de Execução	Seleção de casos de uso e interpretação da rede de conceitos resultante
Análise Formal de Conceito de Identificadores	Inspeção dos conceitos resultantes
Detecção de Clones (*Token*, Árvore Sintática Abstrata e Grafo de Dependência de Programa)	Interpretação dos clones descobertos
Análise de *Fan-in*	Seleção de candidatos a partir da lista de métodos ordenada por valor *fan-in*
Análise Dinâmica e Regras de Associação	Seleção de casos de uso e inspeção das regras resultantes
Análise de *Link*	Interpretação dos valores de *centralização*, *espalhamento* e *ranking*
Fatiamento de Programa	Análise de fatias

Tabela 2.4: Pré-condições da implementação de um programa para cada técnica encontrar candidatos a aspectos viáveis.

Técnica	**Pré-condições**
Análise de Padrões Recorrentes (em Rastros de Execução e em Grafo de Fluxo de Controle)	O *CCC* está implementado em métodos que tratam somente dele, e a ordem de chamada dos métodos é sempre a mesma
Análise Formal de Conceito de Rastros de Execução	Existe pelo menos um caso de uso que expõe o *CCC* e outro que não expõe
Análise Formal de Conceito de Identificadores	Os nomes dos métodos que implementam o *CCC* são parecidos
Detecção de Clones (*Token*, Árvore Sintática Abstrata e Grafo de Dependência de Programa)	O *CCC* está implementado pela reutilização de um fragmento de código
Análise de *Fan-in*	O *CCC* está implementado em métodos que tratam somente dele, e que são chamados um elevado número de vezes
Análise Dinâmica e Regras de Associação	O *CCC* está implementado em métodos que tratam somente dele, e que são chamados um elevado número de vezes
Análise de *Link*	O *CCC* está implementado em métodos de uma classe que trata somente de tal interesse, e que é utilizada por várias outras classes
Fatiamento de Programa	-

2.5 Deficiências das Técnicas

Após a avaliação e comparação das técnicas de mineração de aspectos, tendo notado que cada técnica tem suas próprias limitações e fraquezas, Mens et al. (2008) listaram deficiências típicas observadas que, em geral, as técnicas possuem, como segue:

- *Precisão ruim.* Muitas das técnicas de mineração de aspectos apresentam baixa precisão, isto é, a porcentagem de candidatos a aspectos relevantes no conjunto de todos os candidatos relatados por uma dada técnica é relativamente baixa. Embora essa baixa precisão não seja um problema em si, ela implica na tendência das técnicas em retornar uma grande quantidade de falsos positivos, o que pode ser prejudicial à sua escalabilidade, além de poder exigir muito envolvimento do usuário para separar os falsos positivos dos candidatos a aspectos relevantes.

- *Recall ruim. Recall* é a porcentagem de candidatos a aspectos relevantes retornados em relação ao total de aspectos relevantes, mas alguns não retornados. Em outras palavras, *recall* dá uma ideia de quantos falsos negativos permanecem no código.

- *Subjetividade.* Para muitas técnicas de mineração de aspectos existentes, os resultados produzidos apresentam algumas ambiguidades. Dependendo da pessoa e da definição de aspecto que ela usa, pode ocorrer de uma pessoa dizer que algo é um candidato a aspecto enquanto que outra pessoa pode dizer que não é.

- *Escalabilidade.* Uma propriedade importante de qualquer técnica de mineração de aspectos é a sua escalabilidade. Um fator que tem um impacto sobre a escalabilidade é a eficiência do tempo da ferramenta, isto é, o tempo necessário para a ferramenta calcular seus resultados. A maioria das ferramentas não parece ser problemática a esse respeito. Outro fator que contribui para a escalabilidade, no entanto, é a quantidade de envolvimento do usuário requerida para uma dada técnica. Geralmente, o tempo necessário para uma ferramenta de mineração de aspectos calcular seus resultados é negligenciável no que diz respeito ao tempo necessário para um usuário da ferramenta pré-processar a entrada e/ou pós-processar e analisar sua saída.

- *Validação empírica.* É impossível para a disciplina de mineração de aspectos fazer progressos sem a validação empírica dos resultados. No entanto, a validação da qualidade de uma técnica de mineração de aspectos é um problema intrinsecamente difícil. Uma boa validação empírica requer a capacidade de medir a precisão e o *recall* dos resultados, em diferentes níveis de granularidade. A subjetividade da interpretação dos resultados, no entanto, obstrui uma validação empírica: analisando os resultados de uma técnica ou comparando-a com os resultados de outras técnicas torna-se assunto para os pesquisadores realizarem um experimento, limitando a reprodutibilidade dos resultados. Além disso, para demonstrar a escalabilidade da abordagem, estudos de usuários devem ser realizados. Os usuários finais de técnicas de mineração de aspectos devem ser envolvidos, a fim de avaliar a utilidade e usabilidade efetiva de cada técnica proposta.

- *Comparabilidade.* A comparação de resultados de diferentes técnicas é complexa por diferentes razões: diferença na granularidade (nível de detalhe) dos resultados, diferença nos tipos de resultados (algumas vezes são apenas nomes de métodos, fragmentos de código e métodos que estão sendo chamados, por exemplo) e subjetividade na interpretação dos resultados.

- *Agregabilidade.* Com as limitações observadas de algumas técnicas de mineração de aspectos, há um desejo de *combinar* as técnicas de pesquisadores diferentes (Ceccato et al., 2006). No entanto, por razões semelhantes às mencionadas no problema de *comparabilidade*, tal combinação é trabalhosa na prática.

- *CCCs simples não são tão simples.* Em Bruntink et al. (2007), a implementação de rastreamento em um grande caso industrial foi estudada. Embora o rastreamento é tradicionalmente considerado um *CCC* "simples", descobriu-se que os idiomas dos programadores usados para implementar esse interesse apresentou notável variabilidade. Essa variabilidade significativamente dificulta a tarefa de automaticamente minerar instâncias desse interesse a partir do código. Embora parte dessa variabilidade é acidental e devido a erros de digitação ou utilização indevida de idiomas, uma parte significativa da variabilidade voltou a ser essencial. Portanto, as técnicas de mineração de aspectos devem considerar explicitamente essa variabilidade para minerar os candidatos a aspectos relevantes. Se um interesse "trivial", como o rastreamento, possui variabilidade, os interesses mais complexos são, provavelmente, mais trabalhosos de serem investigados.

Ao analisar as deficiências mencionadas, Mens et al. (2008) identificam três causas principais: i) inadequação das técnicas utilizadas para minerar aspectos; ii) falta de definição precisa do que constitui um aspecto e; iii) apresentação inadequada dos resultados das técnicas de mineração de aspectos. Tais causas são descritas a seguir:

- *Técnicas inapropriadas.* A causa de muitas das deficiências observadas é que a maioria das abordagens não são adequadas ao seu propósito. As razões disso, são:

  - *Propósito geral.* As abordagens de mineração de aspectos utilizam técnicas de mineração de propósito geral a fim de identificar *sementes* ou candidatos a aspectos. Consequentemente, tais abordagens podem ser demasiadamente de propósito geral e resultam em um desempenho ruim. Por outro lado, técnicas dedicadas para buscar tipos de *CCCs* particulares podem obter candidatos a aspectos válidos, porque elas podem ser ajustadas às particularidades do tipo de interesse procurado.
  - *Suposições fortes.* Todas as técnicas de mineração de aspectos fazem certas suposições sobre, por exemplo, como os *CCCs* são implementados no código fonte, para identificar grupos de entidades de código fonte que exibem o sintoma suposto. Exemplos de tais sintomas utilizados pelas técnicas são padrões recorrentes de chamadas, duplicação de código e altos valores *fan-in*. Ao se basear nessas suposições, as técnicas tendem a sofrer de dois problemas. Por um lado, algumas técnicas são dependentes da estrutura do código fonte, isto é, elas procuram *CCCs* revelados por uma maneira específica de como o

interesse e seu código base são estruturados. Consequentemente, essas técnicas somente detectam os candidatos a aspecto que correspondam a suposição particular. Por outro lado, as suposições em que se baseia uma técnica sobre a implementação de um *CCC* geralmente não tratam variações na sua implementação. Os *CCCs*, mesmo aparentemente simples, nem sempre são uniformemente implementados. Essa falta de homogeneidade na implementação de candidatos a aspecto afeta o *recall* das técnicas. *CCCs* com desvios mínimos da suposta implementação podem não ser encontrados por uma técnica.

- *Abordagens otimistas.* A maioria das técnicas (se não todas) são otimistas: elas buscam por sintomas no código fonte com base na hipótese de que um fragmento de código é parte de um aspecto ou de um *CCC*, mas não consideram sintomas que suportam a anti-tese de que o fragmento não é parte de um aspecto ou de um *CCC*. Um fragmento de código pode apresentar todos os sintomas que o tornam parecido com um aspecto ou com um *CCC*, mas essa hipótese pode ser invalidada por argumentos contra.
- *Espalhamento versus emaranhamento.* Dois principais sintomas da presença de aspectos são o espalhamento e o emaranhamento. A maioria das técnicas de mineração de aspectos se concentram exclusivamente na detecção de sintomas de espalhamento. Enquanto o espalhamento certamente é um indicador de *CCC*, somente ele não satisfaz a correta identificação dos candidatos a aspectos válidos. Esse problema foi exemplificado pela precisão e pelo *recall* ruins de algumas das técnicas de detecção de clones. Isso estava relacionado com a quantidade de emaranhamento dos interesses. Detectores de clones alcançaram melhor precisão e *recall* para os interesses que exibiram emaranhamento relativamente baixo com outros interesses do que para os interesses que apresentaram alto emaranhamento. Foi concluído que é importante considerar o sintoma de emaranhamento para a mineração de aspectos.
- *Falta de uso de informação semântica.* Embora os sintomas de *CCCs* (como o espalhamento e o emaranhamento) faz com que seja possível identificar potenciais candidatos a aspectos, eles resultam, também, na introdução de precisão e *recall* ruins. Os *CCCs* geralmente podem ser caracterizados por um sintoma particular, mas não necessariamente significa que todas as entidades de código fonte que exibem tal sintoma fazem parte de um *CCC*. Por exemplo, um sintoma de *CCC* é a duplicação de código, e as técnicas que se concentram em encontrar algumas maneiras de duplicação podem positivamente identificar *CCCs*. No entanto, as técnicas também podem identificar interesses que não são *CCC*. Sem conhecimento semântico é difícil decidir como um trecho de código que faz parte de um *CCC* é acoplado à incorporação do código. Informações semânticas sobre esse acoplamento podem ajudar a decidir se o trecho de código é transversal no sentido orientado a aspectos e, assim, representa um candidato potencial a aspecto.

- *Definição imprecisa.* A segunda principal causa de problemas identificados na maioria das técnicas de mineração de aspectos é a falta de uma definição precisa do que constitui um aspecto ou um *CCC*. Sem uma clara e inequívoca definição do que é para ser minerado é difícil definir

e validar técnicas de mineração apropriadas. Enquanto as técnicas podem identificar a manifestação de candidatos a aspectos úteis no código fonte de um sistema, a falta de uma definição mais formal faz com que a interpretação dos resultados obtidos por essas técnicas seja subjetiva para o usuário da técnica. Consequentemente, isso afeta, também, a realização de validação empírica de uma técnica de mineração de aspectos: a validação dos resultados da técnica é impossível se não é claro o que é exatamente que a técnica tenta encontrar.

- *Apresentação inadequada dos resultados.* A última causa de problemas é relacionada com a apresentação utilizada para os resultados de uma técnica de mineração de aspectos. O nível de granularidade em que os resultados de uma técnica são apresentados podem impactar a qualidade e a compreensão dos resultados. Se uma granularidade grosseira é utilizada pode tornar mais difícil para os usuários a discernir se um proposto interesse é de fato um candidato a aspecto válido. Fornecendo demasiados detalhes, por outro lado, pode resultar no sobrecarregamento do usuário pela quantidade de dados que ele precisa processar a fim de filtrar informações relevantes. No entanto, se os resultados do processo de mineração de aspectos precisam servir como entrada de uma etapa de extração, informações suficientemente detalhadas sobre os *join points* e dos seus contextos devem ser retornadas pela técnica. Adicionalmente, não somente o nível de granularidade, mas também a maneira como os resultados são apresentados ao usuário (nomes de métodos, chamados e chamadores, fragmentos de códigos compartilhados, por exemplo) pode ser uma causa de ambiguidade, porque induz a mente do usuário a determinada interpretação. Finalmente, não existe um formato padrão para a apresentação dos resultados das técnicas de mineração de aspectos, o que torna difícil comparar ou combinar as diferentes técnicas.

Na Tabela 2.5 é sumarizado o impacto das diferentes causas identificadas com as deficiências listadas. O sinal "-" indica um impacto negativo, o sinal "(-)" indica um impacto negativo menos direto e o sinal "(+)" indica um impacto positivo.

Tabela 2.5: Deficiências das técnicas de mineração de aspectos e suas causas (Adaptado de Mens et al. (2008)).

Causa / **Deficiência**	**Técnicas inapropriadas**					**Definição imprecisa**	**Apres. inadeq. dos resultados**
	Propósito geral	Suposições fortes	Abordagens otimistas	Falta atenção p/ emara.	Falta de uso de inf. sem.		
Precisão ruim	-	(+)	-	-	-	-	-
Recall ruim	(+)	-	(+)	-	-	-	
Subjetividade					-	-	-
Escalabilidade	(-)	(+)	(-)	(-)	(+) (-)	-	(-)
Validação empírica						-	-
Comparabilidade						-	-
Agregabilidade						-	-
CCCs simples não são tão simples		-	-	(-)	-	-	

2.6 Considerações Finais

Para a introdução de aspectos em um sistema orientado a objetos é necessário minerar aspectos, identificando candidatos a *CCCs* no código de sistemas de software. Diversas técnicas têm sido propostas para a mineração de aspectos. Por meio de avaliação e comparação, foi observada uma grande variação em tais técnicas, dependendo do tipo de dado analisado, do tipo de análise realizada, da granularidade dos resultados relatados, das suposições que as técnicas fazem sobre como o programa sob analise é implementado, dos sintomas de *CCCs* que são procurados e do tipo de envolvimento do usuário requerido.

Por meio da avaliação e comparação, Mens et al. (2008) notaram que cada técnica tem suas próprias limitações, e listaram as deficiências típicas observados que, em geral, as técnicas possuem. Além disso, os autores identificaram e listaram as principais causas dessas deficiências, sendo uma delas a *apresentação inadequada dos resultados*.

Nesse contexto, abordagens e ferramentas para lidar com as deficiências das técnicas de mineração de aspectos, para que a refatoração para aspectos seja facilitada, podem ser úteis. Assim, a fim de melhorar a apresentação dos resultados obtidos por técnicas de mineração de aspectos, a utilização de *Visualização de Software* pode ser vantajosa, tanto para a interpretação e análise dos resultados, como para a comparação de resultados de diferentes técnicas.

CAPÍTULO 3

Visualização de Informação e de Software

Analisar e explorar grandes volumes de dados tornou-se cada vez mais difícil. *Visualização de Informação* e *Mineração de Dados* podem ajudar a lidar com tal problema (Keim et al., 2005). A Mineração de Dados pode ser definida como o processo de pesquisa e de análise de dados com o objetivo de encontrar informações implícitas, mas potencialmente úteis (Frawley et al., 1991). Entretanto, para a Mineração de Dados ser efetiva, é importante incluir o humano no processo de exploração dos dados e combinar a flexibilidade, criatividade, conhecimento geral e habilidades de percepção do ser humano com a enorme capacidade de armazenamento e processamento dos computadores (Keim et al., 2005). A Mineração *Visual* de Dados utiliza a *Visualização* como um canal de comunicação entre computador e humano para apoiar a identificação de novos padrões (Ankerst, 2000). A *Visualização* é definida como o uso interativo de representações visuais de dados para amplificar a cognição (Card et al., 1999), sendo que, nesse contexto, entende-se por cognição a aquisição ou o uso de conhecimento por uma pessoa (Garcia, 2006).

O emprego de Visualização no processo de Mineração de Dados introduz algumas possibilidades, como: percepção de propriedades não antecipadas, essencial para o descobrimento de novos padrões; percepção simultânea das características dos dados em grande e pequena escala; apoio a processos de formação de hipóteses; apoio a tarefas de pré-processamento dos dados, como detecção de problemas, limpeza, seleção, dentre outras (Garcia, 2006).

Diversas técnicas de Visualização de Informação têm sido propostas para apoiar a exploração de dados, bem como técnicas de interação para tornar tal exploração eficaz (as técnicas apresentadas nas seções 3.1 e 3.2 são utilizadas no desenvolvimento deste trabalho). No entanto, as técnicas de visualização possuem limitações e, assim, pode ser vantajoso utilizar diferentes técnicas em conjunto. Segundo Keim e Kriegel (1996), múltiplas visões permitem explorar os pontos fortes e minimizar o efeito dos pontos fracos das técnicas de visualização, e podem fornecer mais informações do que cada técnica individual (Seção 3.3). Adicionalmente, a visualização tem sido utilizada para apresentar artefatos de software, assunto apresentado na Seção 3.4.

3.1 Técnicas de Visualização de Informação

Existem diversas técnicas de visualização, utilizadas para apresentar diferentes tipos de dados e para diferentes propósitos. Neste trabalho são descritas as técnicas *Treemap* (Shneiderman, 1990, Johnson e Shneiderman, 1991) e *Hyperbolic Browser* (Lamping et al., 1995, Lamping e Rao, 1996), que são utilizadas no desenvolvimento do projeto.

A técnica *Treemap* (Shneiderman, 1990, Johnson e Shneiderman, 1991) mapeia estruturas hierárquicas para uma projeção 2D retangular em um modo de preenchimento de espaço (*space-filling*), sendo o espaço designado à projeção particionado em uma coleção de regiões retangulares delimitadoras e aninhadas que representam a estrutura de árvore. Controle interativo permite aos usuários especificar a apresentação de informações estruturais (limites de profundidade, etc.) e conteúdo (propriedades visuais, como mapeamentos de cores) (Shneiderman, 1990, Johnson e Shneiderman, 1991). Na Figura 3.1 é ilustrada a ferramenta *TreeViz* (Human Computer Interaction Lab, 2013) na qual foi implementada a técnica *Treemap*, mostrando todos os arquivos e diretórios em um disco rígido. Cada arquivo aparece como um retângulo cujo tamanho é proporcional ao tamanho do arquivo, permitindo aos usuários localizar arquivos grandes em qualquer nível da hierarquia. *TreeViz* utiliza coloração para mostrar o tipo de arquivo, por exemplo texto, imagem e aplicação. Ao apontar e clicar em um retângulo, os usuários podem visualizar informações detalhadas sobre os nós, como nome do arquivo, caminho e data de criação (Human Computer Interaction Lab, 2013).

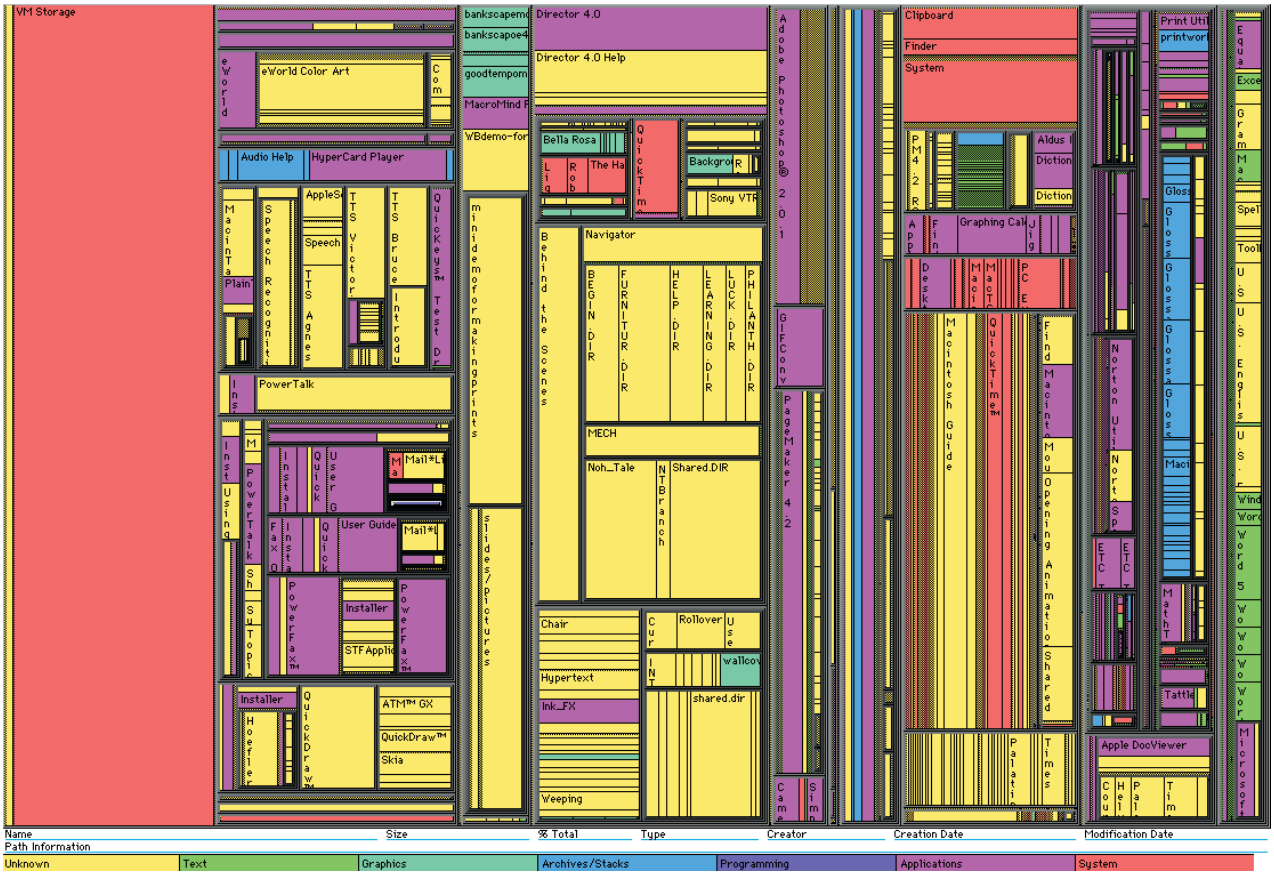


Figura 3.1: Instantâneo da interface da *TreeViz* mostrando uma hierarquia de arquivos (Human Computer Interaction Lab, 2013).

Hyperbolic Browser (Lamping et al., 1995, Lamping e Rao, 1996) é uma técnica *foco + contexto* para visualização e manipulação de grandes árvores hierarquicas ou grafos. Algumas estruturas são mais importantes, chamadas de "regiões de interesse" ou "regiões de foco", enquanto outras estruturas são menos importantes e servem apenas como o "contexto". Técnicas *foco + contexto* tem o objetivo de revelar as regiões de interesse preservando o contexto (Wu et al., 2006).

Na Figura 3.2 é ilustrado o *Hyperbolic Browser*, em que inicialmente é exibida a raiz de uma árvore no centro (foco). Pode-se perceber duas propriedades salientes: os componentes diminuem de tamanho conforme aumentam a distância do nó em foco (distorção *fisheye*), e há um crescimento exponencial do número de componentes à medida que aumenta o raio da projeção. Assim, o contexto inclui várias gerações de nós, tornando mais fácil para o usuário explorar a hierarquia. Adicionalmente, foram desenvolvidos procedimentos eficazes para manipular o foco usando eventos (clique do *mouse*) e transições animadas para o redirecionamento contínuo do foco. Na Figura 3.3 são mostrados quatro quadros que ilustram as transições animadas quando o foco é alterado. Nesse exemplo de mudança de foco, o nó atual em foco está destacado em amarelo ("Allaire") e o nó selecionado pelo usuário para ser o novo nó em foco está destacado em vermelho ("Hicks").

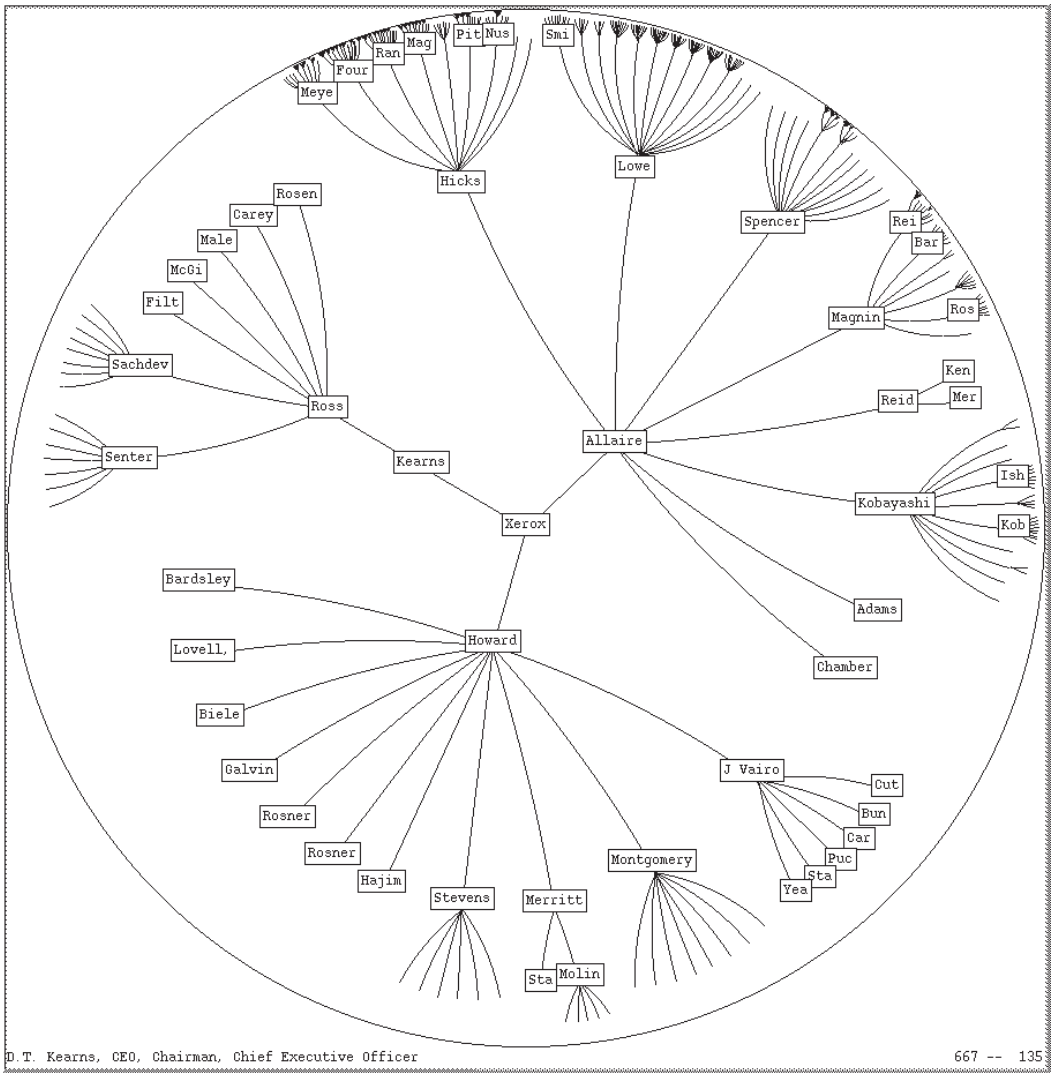


Figura 3.2: *Hyperbolic Browser* (Lamping et al., 1995).

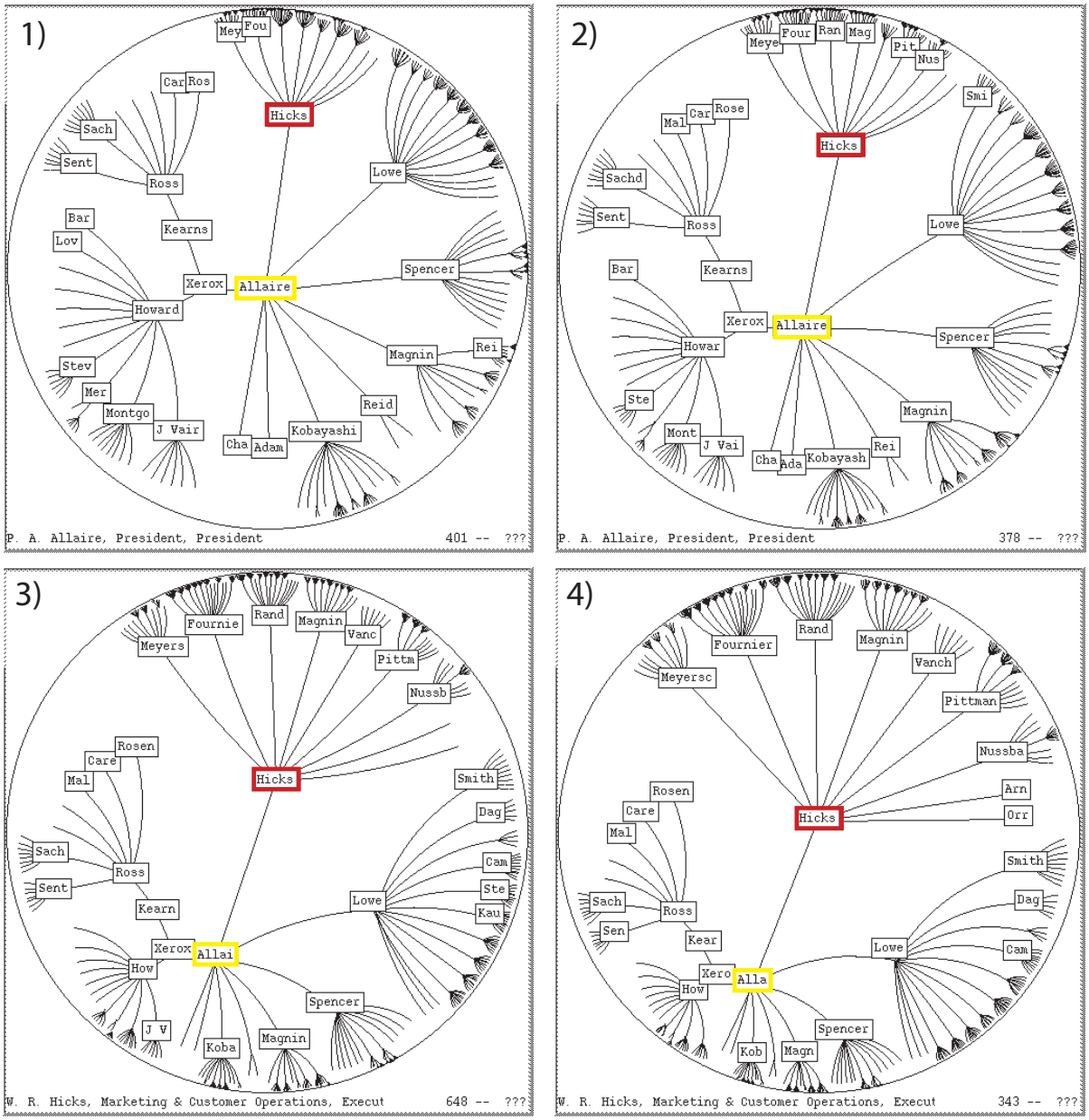


Figura 3.3: *Hyperbolic Browser* – Séries de quadros que ilustram a animação quando da mudança de foco (Lamping et al., 1995).

3.2 Técnicas de Interação

Em adicional às técnicas de visualização, para uma exploração de dados eficaz é necessário utilizar *técnicas de interação*. Técnicas de interação permitem o analista de dados interagir com as visualizações e alterá-las dinamicamente de acordo com os objetivos da exploração. Além disso, tais técnicas tornam possível relacionar e combinar múltiplas visualizações independentes. Alguns exemplos de efeitos que se pode ter em projeções utilizando técnicas de interação, são: ajuste do nível de detalhe em parte ou na totalidade da visualização; modificação do mapeamento para enfatizar algum subconjunto dos dados; isolamento de um subconjunto dos dados exibidos para operações como destaque, filtragem e análise quantitativa, por meio de seleção direta na visualização (manipulação direta) ou por meio de caixas de diálogo ou de mecanismos de consulta (manipulação indireta) (Keim

et al., 2005). Algumas técnicas de interação são descritas a seguir.

3.2.1 Filtragem Interativa

Ao explorar grandes conjuntos de dados é importante particionar interativamente o conjunto de dados em segmentos e focar em subconjuntos interessantes. Isso pode ser feito por meio de *filtragem interativa*, podendo ser por seleção direta do subconjunto desejado (*browsing*) ou por uma especificação de propriedades do subconjunto desejado (*querying*) (Keim et al., 2005).

3.2.2 Zoom Interativo

Zoom é uma técnica bem conhecida e largamente utilizada em diversas aplicações. Ao lidar com grandes quantidades de dados, é importante apresentar os dados de uma maneira altamente comprimida para proporcionar uma visão geral dos dados, mas também permitir uma visualização variável dos dados em diferentes resoluções. *Zoom* não significa apenas exibir os objetos maiores, mas também significa que a representação de dados pode mudar automaticamente para apresentar mais detalhes sobre os níveis superiores de *zoom*. Os objetos podem, por exemplo, ser representados como *pixels* únicos em um nível de *zoom* baixo, como ícones em um nível de *zoom* intermediário, e como objetos rotulados em alta resolução (Keim et al., 2005).

3.2.3 Distorção Interativa

Distorção interativa é uma técnica de modificação de visão que apoia o processo de exploração de dados preservando uma visão geral dos dados durante operações *drill-down* (subseção 3.3.4). A ideia básica é mostrar partes dos dados com um alto nível de detalhe, enquanto outros são mostrados com um menor nível de detalhe. Técnicas de distorção populares são as distorções hiperbólicas e esféricas, que são frequentemente utilizadas em hierarquias ou grafos, mas também podem ser aplicadas em qualquer outra técnica de visualização (Keim et al., 2005).

3.2.4 Seleção & Ligação Interativa

A operação de *seleção* (*brushing*) se refere ao processo de selecionar um subconjunto de registros apontando diretamente os seus elementos gráficos, por exemplo, com o *mouse* (Garcia, 2006). Quando a seleção é combinada com *ligação* (*linking*), os elementos selecionados são destacados em outras visualizações, o que torna possível detectar dependências e correlações (Keim et al., 2005). Esse assunto é apresentado na próxima seção.

3.3 Múltiplas Visões e Técnicas de Coordenação

As diferentes técnicas de visualização apresentam vantagens e desvantagens (Keim et al., 2005). Para amenizar as deficiências individuais de cada técnica, outras podem ser adicionadas ao processo de exploração. A adição de diferentes tipos de técnicas pode sobrecarregar o usuário na adaptação do contexto de cada uma. Então, para facilitar a associação que o usuário tem que fazer entre elas, *técnicas de coordenação* podem ser utilizadas para ajudar a conectar as diferentes visões[1]. A área que trata deste problema é conhecida como *Múltiplas Visões Coordenadas* ou *Linked Views*. Geralmente,

[1] Visão é a representação visual de um conjunto de dados por meio de uma técnica de visualização (Eler, 2011).

sistemas que utilizam múltiplas visões coordenadas possuem mais de uma técnica de visualização para explorar um mesmo conjunto de dados, conjuntos de dados diferentes ou diferentes subconjuntos de um único conjunto de dados (Eler, 2011). As múltiplas visões podem ser coordenadas de diversas maneiras. Algumas técnicas de coordenação são descritas nas próximas subseções.

3.3.1 *Brushing-and-Linking*

Brushing-and-Linking é uma técnica de coordenação utilizada quando são exibidos itens de um conjunto de dados em múltiplas visões. Quando ocorre a seleção de um item em uma visão, os itens correspondentes são automaticamente destacados ou selecionados em outras visões. Normalmente essa técnica é utilizada para identificar itens quando um conjunto de itens é exibido em diferentes visões para diferentes contextos. Por exemplo, na Figura 3.4 são mostrados dados censitários projetados usando *Spotfire* (Ahlberg e Wistrand, 1995), um pacote de análise de dados comerciais. Selecionando os estados com baixos percentuais de pessoas com ensino médio e curso superior completo no gráfico da esquerda, é revelado que os estados também têm baixa renda e elevados níveis de desemprego no gráfico da direita (North e Shneiderman, 2000, North, 2000).

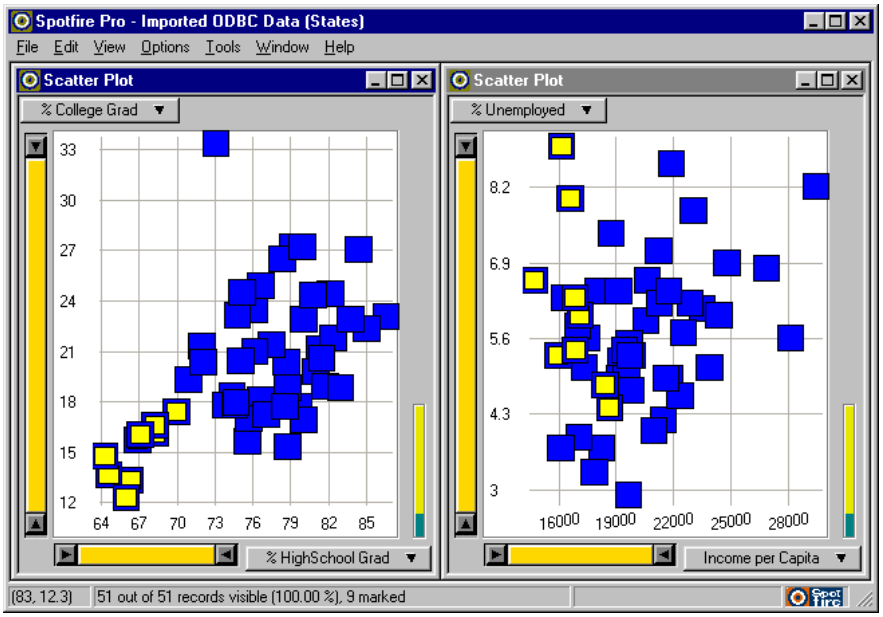


Figura 3.4: *Brushing-and-linking* no *Spotfire* (North, 2000).

3.3.2 *Overview and Detail*

Overview and Detail é utilizada quando itens são representados visualmente em um tamanho menor em uma visão geral do que na exibição de detalhes. Quando ocorre a seleção de um item na visão geral, são mostrados detalhes do item selecionado na visão de detalhes, permitindo acesso direto aos detalhes por meio de um contexto. Por exemplo, *web designers* normalmente adicionam uma tabela de conteúdo para uma página *web* grande. Os usuários podem selecionar o título de uma seção para rolar a página *web* imediatamente para a seção selecionada. Na Figura 3.5, por exemplo, foi selecionada a seção "Financial Information" do catálogo de graduação (North e Shneiderman, 2000, North, 2000).

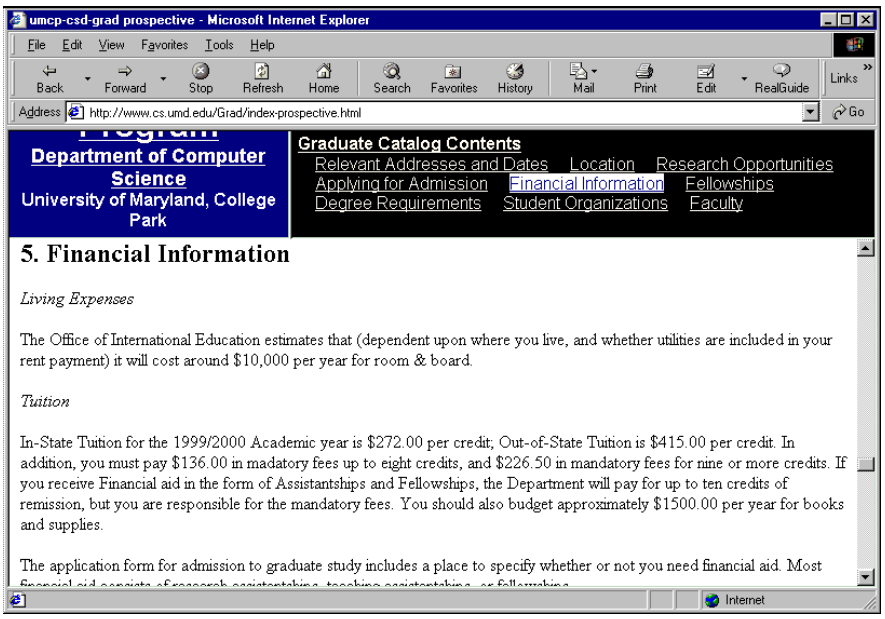


Figura 3.5: *Overview and detail* com quadros da web (North, 2000).

3.3.3 *Synchronized Scrolling*

Synchronized Scrolling é uma técnica utilizada entre visões que possuem barra de rolagem, de modo que quando o usuário percorre uma lista de itens em uma visão, os itens correspondentes são percorridos em uma outra visão. Na Figura 3.6 os usuários do software *Logos Bible* (Logos Research Systems, 1993) podem simultaneamente percorrer várias traduções e comentários da Bíblia por capítulo e versículo. Isso facilita comparações ou examinação de dados a partir de múltiplas visões (North e Shneiderman, 2000, North, 2000).

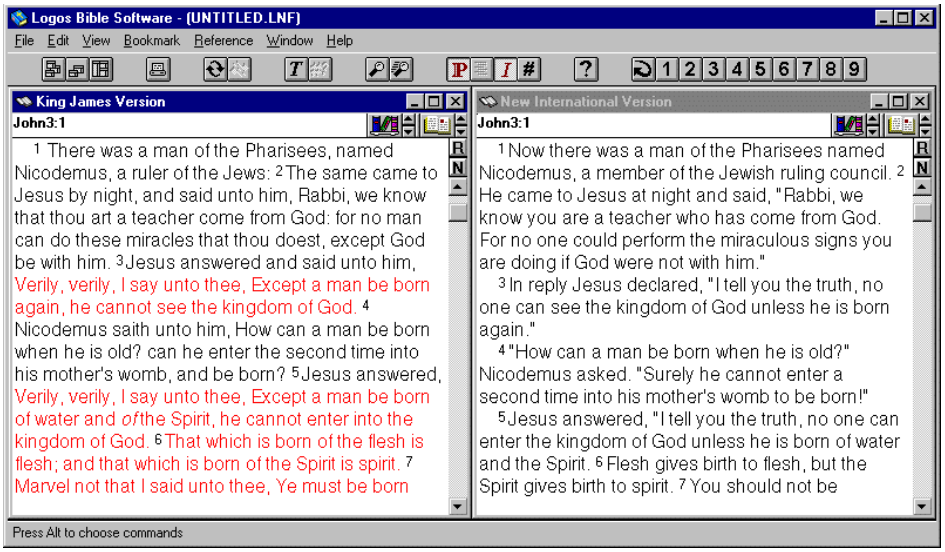


Figura 3.6: *Synchronized scrolling* com o software *Logos Bible* (North, 2000).

3.3.4 *Drill Down*

Drill Down é uma técnica que permite a navegação de camadas sucessivas de dados hierárquicos, de modo que quando ocorrer a seleção de um item pai em uma visão, os itens filhos são carregados

em outra visão. Isso permite explorar os dados em escala muito grande por meio da exibição de agregados em uma visão e o conteúdo de um agregado selecionado em uma outra visão. Na Figura 3.7 é mostrado um mapa de registros de incidente a partir de *Baltimore Beltway*. O tamanho dos marcadores depende do número de incidentes e a cor depende da distância de uma unidade de resposta. Com os agregados em um mapa, é fácil identificar onde a maioria dos incidentes ocorreu. Quando o usuário seleciona um agregado, os incidentes são mostrados em um mapa detalhado com a localização de cada incidente (Fredrikson et al., 1999, North e Shneiderman, 2000, North, 2000).

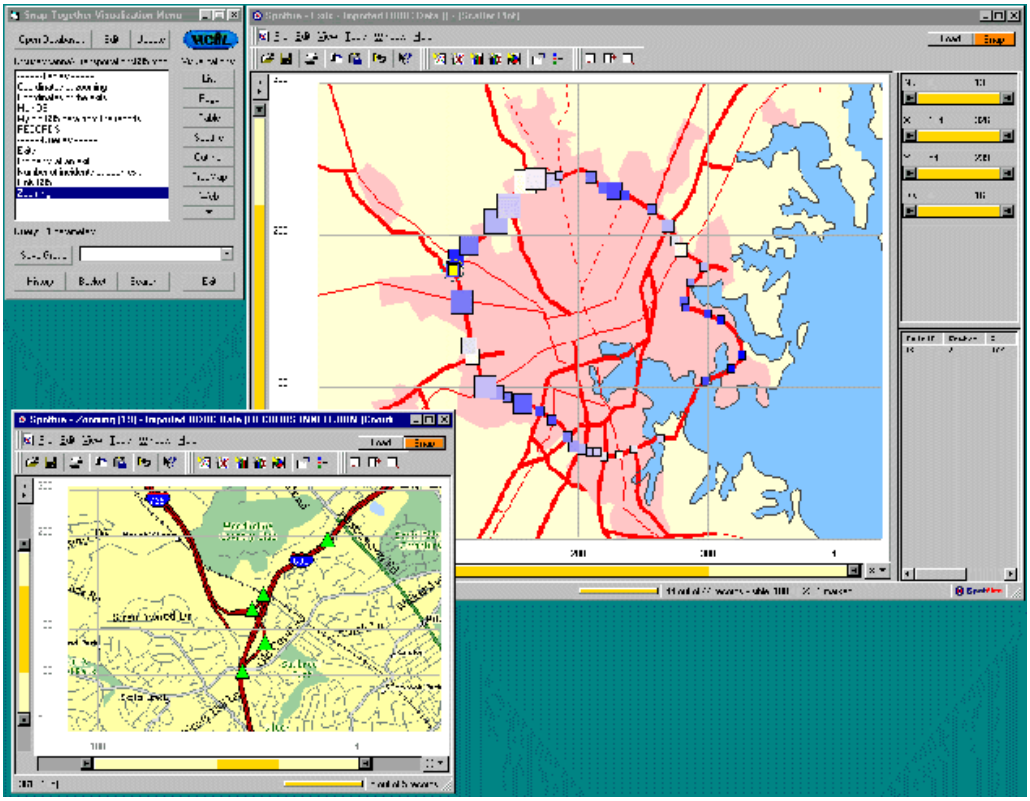


Figura 3.7: Visualização geográfica de incidentes ocorridos (Fredrikson et al., 1999).

3.4 Visualização de Software

Software é intangível, não têm forma física ou tamanho (Ball e Eick, 1996). A *Visualização de Software* (VS) é o uso de tecnologia de computação gráfica e de interação humana para facilitar a compreensão e o uso efetivo de software de computador. A VS pode ser dividida em *Visualização de Programa* (VP) e *Visualização de Algoritmo* (VA). VP é a visualização de código ou estruturas de dados de programa e VA é a visualização de um nível alto de abstração o qual descreve o software. Ambos os tipos de visualização podem ser estática ou dinâmica, isto é, a visualização não muda enquanto o algoritmo é executado ou a visualização muda para refletir os eventos que ocorrem durante a execução (Stasko et al., 1998).

Ferramentas de visualização de software utilizam técnicas gráficas para tornar o software visível por meio de metáforas visuais que representam programas, artefatos, estrutura e comportamento de programa, por exemplo. O objetivo essencial é que as representações visuais podem tornar o processo de compreensão de software mais fácil (Ball e Eick, 1996). Na próxima subseção são descritas

algumas ferramentas de visualização de software.

3.4.1 Ferramentas de Visualização de Software

Seesoft (Eick et al., 1992) é uma ferramenta de visualização de software que implementa quatro ideias-chave: representação reduzida do código, coloração por estatística, manipulação direta e capacidade de leitura do código. A representação reduzida é realizada pela exibição de arquivos como colunas e linhas de código como linhas finas. A cor de cada linha fina é determinada por uma estatística associada com a linha de código que a linha fina representa, como por exemplo, a data que a linha de código foi criada (idade do código). Usando técnicas de manipulação direta, em particular a atualização da tela em tempo real em resposta de ações do *mouse*, o usuário manipula a exibição para encontrar padrões de seu interesse no código. Além disso, o usuário pode abrir uma janela de leitura de código, que exibe o código correspondente às linhas marcadas por caixas de aumento virtuais sobre a apresentação reduzida. Assim, o usuário possui um visão geral das estatísticas (representação reduzida) e uma visão detalhada (trechos de código) de partes do programa em análise (Eick et al., 1992).

TARANTULA (Jones et al., 2002) é uma ferramenta de visualização de sofware cujo objetivo é apoiar a tarefa de localização de defeitos. A ferramenta implementa uma técnica baseada em cores para mapear visualmente a participação de cada instrução do programa no resultado da execução de um conjunto de testes. Com base nesse mapeamento visual, um usuário pode inspecionar as instruções do programa, identificar instruções envolvidos em falhas e localizar instruções potencialmente defeituosas.

CODECRAWLER (Lanza et al., 2005) é uma ferramenta de visualização de software independente de linguagem de programação. O CODECRAWLER oferece a vantagem de ter múltiplas visões da mesma arquitetura de software. As visualizações são baseadas nas visões polimétricas descritas por Lanza (2003), Lanza e Ducasse (2003), em que as entidades do código fonte (classes, métodos, etc.) são representadas como nós e os seus relacionamentos como arestas entre os nós. O tamanho, a cor e a posição dos nós são usados para representar as métricas de interesse do sistema de software. A multiplicidade de visões tem como objetivo descobrir os diferentes aspectos do projeto arquitetural e enfatizar métricas específicas no sistema de software.

Visualiser[2] é um *plug-in* extensível do *Eclipse*[3] que pode ser usado para visualizar qualquer conjunto de dados na representação de barras e listras. Inicialmente, o *Visualiser* era nomeado como *Aspect Visualiser*, parte do *AJDT* (*AspectJ Development Tools*)[4], originalmente criado para visualizar como aspectos afetam classes em um projeto. Desde então, o *Aspect Visualiser* tem sido extraído para se tornar um *plug-in* independente do *AJDT*, a fim de que outros tipos de informações possam ser visualizadas (The Eclipse Foundation, 2012). Um exemplo de uso do *Visualiser* é a visualização de detecção de clones (Tairas et al., 2006).

Asbro (Pfeiffer e Gurd, 2006) é um protótipo de ferramenta de visualização de software orientado a aspectos integrado no *Eclipse IDE*, que pode ser usado juntamente com o *AJDT*. A técnica *Treemap*

[2] `http://www.eclipse.org/ajdt/visualiser/`
[3] `http://www.eclipse.org`
[4] `http://www.eclipse.org/ajdt/`

é implementada tornando estruturas orientadas a aspectos explícitas, com a principal vantagem de ser capaz de lidar com um grande número de elementos de uma maneira estruturada, o que ajuda a obter uma melhor compreensão do código fonte orientado a aspectos, mostrando os pacotes e classes que são entrecortadas e por quais aspectos.

$SoftViz_{OAH}$ (d'Arce et al., 2012) é uma ferramenta de visualização de software na qual foi implementada uma abordagem de múltiplas visões coordenadas para visualizar programas orientados a aspectos. A abordagem emprega três representações visuais e uma lista de exibição – *Apresentação Estrutural*, que utiliza a técnica *Treemap*, têm como objetivo mostrar o código do programa hierarquicamente organizado em pacotes, classes, métodos, aspectos e *advices*; *Apresentação Inter-Unidades*, que utiliza uma projeção *Hiperbólica*, têm como objetivo mostrar o entrecorte de aspectos entre métodos e *advices*; *Apresentação Intra-Método*, que utiliza um *Grafo de Fluxo de Controle*, têm como objetivo mostrar o comportamento intra-método depois do processo de *weaving*; lista de exibição de métodos/*advices*, que mostra os métodos e os *advices* de classes ou aspectos na exploração visual. As representações visuais e a lista de exibição são coordenadas, o que torna possível destacar elementos selecionados em diferentes níveis de detalhe, permitindo que o usuário colete informações sobre aspectos e seu espalhamento no código, isto é, como aspectos entrecortam as estruturas do programa. Adicionalmente, a $SoftViz_{OAH}$ é capaz de mostrar resultados de casos de teste juntamente com as representações visuais, que permite analisar como os casos de teste passam pelo código e sua cobertura.

3.5 Considerações Finais

Analisar e compreender grandes volumes de dados para obter informações são atividades difíceis, que podem ser beneficiadas com o uso de visualização. Especificamente, a Visualização de Software tem sido utilizada em diversas áreas e atividades em Engenharia de Software. Uma de suas aplicações é na compreensão de programa e de software, que apoia diversas outras atividades, como manutenção de software e refatoração de código. Assim, o uso de visualização de software para análise e exploração de características de software (como métricas, estrutura e comportamento), juntamente com resultados obtidos por técnicas de mineração de aspectos (assunto discutido no Capítulo 2), pode ajudar na identificação de interesses transversais e na refatoração deles para aspectos.

As características de software possuem níveis de abstração diferentes, assim como os resultados obtidos por técnicas de mineração de aspectos. Então, múltiplas visões podem ser utilizadas, sendo cada visão para apresentar uma característica diferente do software, juntamente com resultados de mineração de aspecto. As técnicas de visualização *Treemap* e *Hyperbolic Browser* apresentadas neste capítulo, bem como as técnicas de interação, são utilizadas no desenvolvimento deste trabalho. Adicionalmente, as múltiplas visões propostas são coordenadas, utilizando as técnicas de coordenação descritas neste capítulo, permitindo uma exploração visual de características de programa de software e resultados de técnicas de mineração de aspectos em diferentes níveis de detalhes.

CAPÍTULO 4

O Modelo de Coordenação Proposto e a Ferramenta $SoftVis_{4CA}$

Uma dificuldade em minerar aspectos é a análise e a interpretação dos resultados gerados por técnicas de mineração de aspectos. Mens et al. (2008) justificaram tal afirmação com base no nível de granularidade em que os resultados de uma técnica são apresentados (podendo impactar na compreensão) e na maneira como os resultados são apresentados (podendo haver ambiguidades). Em relação à granularidade, se é grosseira, pode tornar mais difícil para os usuários a discernir se um candidato a aspectos é um candidato válido; e se é muito detalhada, pode resultar no sobrecarregamento do usuário para filtrar informações relevantes. Entretanto, os resultados relatados por técnicas de mineração de aspectos devem conter informações suficientemente detalhadas, pois elas não são utilizadas apenas para a confirmação de interesses transversais no código fonte, mas também para a refatoração para aspectos, no caso do usuário desejar introduzir aspectos. Então, informações sobre os *join points* e seus contextos devem ser retornadas e apresentados pelas técnicas, também. Adicionalmente, Mens et al. (2008) também relatam ser difícil comparar ou combinar as diferentes técnicas por não existir um formato padrão para a apresentação dos resultados delas.

Nesse contexto, abordagens para lidar com as dificuldades e limitações relacionadas com a apresentação inadequada dos resultados das técnicas propostas para minerar aspectos podem ser úteis. O uso de Visualização de Software é uma alternativa que pode ajudar na análise e na interpretação dos resultados de técnicas de mineração de aspectos, apresentando, também, características de software que podem ajudar o usuário a compreender o software. Neste capítulo é apresentada uma abordagem visual (com foco em questões de implementação) para apoiar a mineração de aspectos de programas orientados a objetos implementados em Java.

A abordagem proposta inclui seis representações visuais coordenadas, sendo cada uma delas para apresentar uma perspectiva diferente de programa. Duas técnicas de mineração de aspectos foram escolhidas para serem implementadas. Foi escolhido *fatiamento de programa* porque o seu uso foi proposto na literatura não somente para identificar interesses transversais, mas também para apoiar a

refatoração para aspectos. No entanto, a proposta de seu uso por Katti et al. (2012) consiste em fatiar um programa inteiro e, então, o usuário deve analisar todas as fatias obtidas em busca de fatias que implementam interesses transversais. Dessa maneira, o esforço por parte do usuário é muito grande. A fim de evitar tal esforço, foi escolhida também a *análise de fan-in*, com a ideia de que em um alto nível o usuário possa criar hipóteses sobre unidades que implementam interesses transversais visualizando resultados da métrica *fan-in* e, a partir disso, refinar a sua hipótese visualizando só, e somente só, fatias referentes à tais unidades. Nosso foco é investigar se a visualização contribui na compreensão de programas por meio dos resultados gerados usando as técnicas fatiamento de programa e análise de *fan-in*, propostas para minerar aspectos, de maneira complementar. Uma ferramenta de visualização de software, nomeada $SoftVis_{4CA}$ (*Software Visualization for Code Analysis*), foi desenvolvida para apoiar a abordagem visual proposta.

4.1 A Ferramenta $SoftVis_{4CA}$

A abordagem visual proposta foi implementada em uma ferramenta de Visualização de Software *desktop*, nomeada $SoftVis_{4CA}$. A ferramenta $SoftVis_{4CA}$ foi desenvolvida usando as linguagens de programação *Java* e *AspectJ*.

Na Figura 4.1 é apresentada a arquitetura da ferramenta, organizada em três camadas: *Camada de Dados*, *Camada de Controle* e *Camada de Visualização*. Usando como entrada um arquivo `.jar` contendo a implementação de um programa desenvolvido em *Java* e casos de teste criados usando *JUnit*, na *Camada de Dados* são realizadas as análises estática e dinâmica[1], criando: (i) um projeto com todos os conjuntos de dados necessários para gerar e coordenar as representações visuais em que, inclusive, fazem parte desses conjuntos de dados os resultados da métrica *fan-in* e as representações intermediárias para o cálculo de fatias; (ii) a representação das unidades de programa (pacotes, classes, métodos e casos de teste) e suas características lidas (atributos e variáveis locais). Os casos de teste são utilizados apenas para a execução do programa, para que seja possível obter informações sobre instruções executadas para o cálculo de fatias dinâmicas.

Na *Camada de Visualização*, os conjuntos de dados obtidos na *Camada de Dados* são mapeados para atributos visuais e, a partir desses atributos, são providas as representações visuais. Na *Camada de Controle* é realizada a coordenação das representações visuais, isto é, por meio do recebimento de eventos da interação do usuário em alguma representação visual, são obtidos dados relacionados ao item visual com o qual houve interação, pela consulta aos dados da *Camada de Dados*, e é disparado um evento para a *Camada de Visualização* informando o novo estado da representação visual, como itens destacados, novo foco e nova posição das informações na apresentação. Assim, é permitida uma exploração visual de programa em diferentes níveis de detalhes.

Nas próximas subseções são apresentados detalhes de cada módulo de implementação da ferramenta $SoftVis_{4CA}$. Para facilitar a compreensão, diagramas parciais de classes e/ou pacotes são ilustrados.

[1] A implementação das análises estática e dinâmica foi reutilizada e adaptada de Martins (2007).

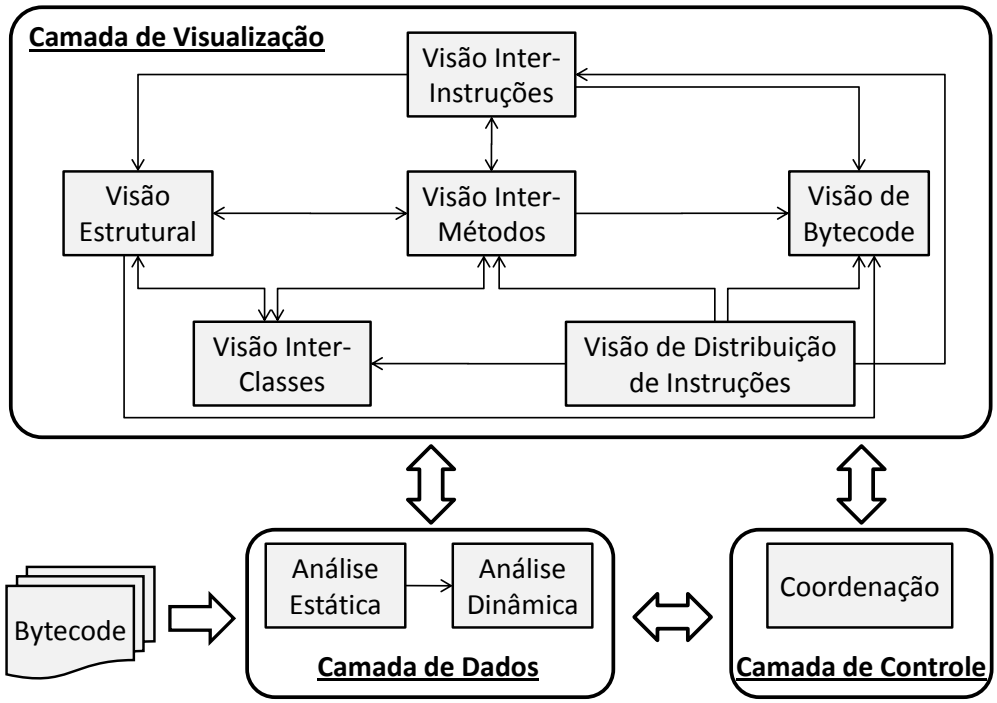


Figura 4.1: Arquitetura da ferramenta $SoftVis_{4CA}$.

4.2 Análise Estática

A *Análise Estática* é responsável por três atividades, em que ocorrem na seguinte ordem:

1. Leitura de classes e de casos de teste de programa a partir de seus códigos bytecode, para a obtenção de informações do código e para a construção de suas representações;

2. Instrumentação de código bytecode de classes, para que seja possível o monitoramento da execução de casos de teste pela *Análise Dinâmica*;

3. Carregamento de instruções bytecode de classes e de casos de teste na máquina virtual, para que seja possível a execução de casos de teste pela *Análise Dinâmica*.

A leitura e a instrumentação de código bytecode foram realizadas usando o *framework* de análise e de manipulação de bytecode ASM[2].

O diagrama de classes do pacote `CodeAnalysis.Static`, no qual contém a implementação da *Análise Estática*, é apresentado na Figura 4.2. A classe `StaticAnalysis` possui os métodos `readClass` e `readTestCase` que são responsáveis por: (i) ler classes e casos de testes, respectivamente; (ii) criar tais no modelo de entidades para representá-los; (iii) e adicioná-los ao projeto. Além disso, são criados, também, grafos em nível de instrução para cada método do programa. Tais atividades, que ocorrem durante a leitura de código bytecode, são descritas nas próximas subseções.

[2] `http://asm.objectweb.org/`

Figura 4.2: Diagrama de Classes do Pacote `CodeAnalysis.Static`.

4.2.1 Modelo de Entidades

Durante a leitura de instruções bytecode na *Análise Estática*, as unidades de programa e as características de unidades lidas são representadas por objetos específicos, instanciados com base no modelo de entidades definido. O diagrama de classes do pacote `Entities`, o qual contém a implementação do modelo de entidades, é apresentado na Figura 4.3. As unidades de programa, e as classes que as representam, são:

- *Pacote*, representado por objeto da classe `Package`, que contém *classes* e *casos de teste*;
- *Classe*, representada por objeto da classe `Class` se é uma classe externa e `InnerClass` se é uma classe interna. A identificação de uma classe é pelo seu nome completo, que é composto pelo nome do pacote ao qual pertence e pelo seu nome. Uma classe é composta por *classes internas*, *métodos* e *atributos* (representados por objetos da classe `Field`). Dentre outras informações da classe, é guardado o código da mesma para ser carregado na máquina virtual para posterior execução;
- *Caso de teste*, representado por objeto da classe `TestCase`. De maneira similar à *classe*, a identificação de um caso de teste é pelo seu nome completo. Um caso de teste é composto por *métodos* e *testes estruturais* criados. Além disso, também é guardado o código do caso de teste;
- *Método*, representado por objeto da classe `Method`. A identificação de um método é pelo nome da classe a qual pertence, pelo seu nome e seu descritor. Cada método possui um `DUG`, construído pela *Análise Estática*, e suas *variáveis locais*, representadas por objetos da classe `LocalVariable`.

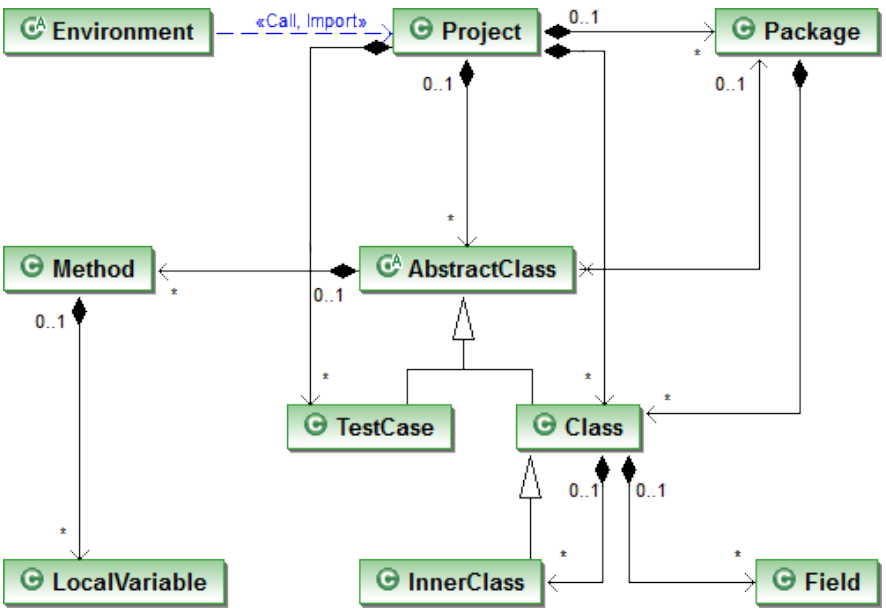


Figura 4.3: Diagrama de Classes do Pacote `Entities`.

Os objetos instanciados para representar classes e casos de teste são como itens de um objeto *projeto*, que é uma instância da classe `Project`, e são carregadas na máquina virtual. Diversos

projetos podem ser criados na ferramenta, os quais são adicionados ao ambiente da $SoftVis_{4CA}$, representado pela classe `Environment`. A classe `Environment` é utilizada para persistência e para recuperação dos projetos.

A representação a partir do modelo de entidades é utilizada para organizar a ferramenta. Os projetos criados, juntamente com suas unidades, são mostrados em uma árvore de projetos. Além disso, os objetos instanciados a partir do modelo de entidades são `userObjects` de itens visuais das visões geradas pela *Camada de Visualização*, utilizados para gerar e coordenar as visões, juntamente com os conjuntos de dados gerados a partir deles.

4.2.2 Construção de *DUG*s

Ainda durante a leitura da *Análise Estática*, é criado um *Grafo de Definição-Uso* (do inglês *Definition-Use Graph – DUG*) (Rapps e Weyuker, 1985) para cada método de classe. O *DUG* é derivado do *Grafo de Fluxo de Controle* (do inglês *Control Flow Graph – CFG*), contendo os conjuntos de definição e uso de variável nos nós/arestas. O diagrama de pacotes e de classes que contém a implementação da representação de *DUG*s é ilustrado na Figura 4.4.

A classe `DUGBuilder`, do pacote `CodeAnalysis.Static` (vide Figura 4.2), estende a classe `MethodVisitor` do *framework* ASM, interceptando a leitura de instruções bytecode e construindo um *DUG* para cada método, representado pela classe `DUG`, do pacote `ProgramRepresentation.InstructionGraph`. Os vértices do grafo representam instruções bytecode agrupadas de acordo com *labels*[3] encontrados no bytecode, juntamente com as definições e os usos de dados em tais instruções. Tais vértices são instâncias da classe `PhysicalVertex`, do pacote `ProgramRepresentation.InstructionGraph.Vertex`). Além desse tipo de vértice, um *DUG* possui também um vértice de entrada (representado por uma instância da classe `EntryVertex`) e um vértice de saída (representado por uma instância da classe `ExitVertex`). As arestas do grafo representam fluxo de controle, podendo ser fluxo de controle *regular* ou fluxo de controle de *exceção* (representados por instância da classe `RegularEdge` e da classe `ExceptionEdge`, do pacote `ProgramRepresentation.InstructionGraph.Edge`).

Durante a leitura, as instruções lidas são inseridas no bloco atual até que o mesmo termine, e são inseridos no grafo o vértice que representa o bloco atual e as arestas que são adjacentes ao vértice. Segundo Martins (2007), as seguintes instruções ou situações determinam o fim de um bloco de instruções bytecode:

- *Label*: o bloco atual encerra, e se antes do label ocorreu um desvio incondicional ele não se conecta ao próximo bloco; caso contrário ele se conecta;
- *Um desvio incondicional (goto)*: o bloco atual encerra e não se conecta ao próximo, mas se conecta ao bloco cujo label é o alvo do desvio;
- *Um desvio condicional (if)*: o bloco atual encerra e se conecta ao próximo; além disso, ele se conecta com o bloco cujo label é o alvo do desvio;

[3] *Label* é uma instrução bytecode que indica o início de um novo bloco de instruções bytecode.

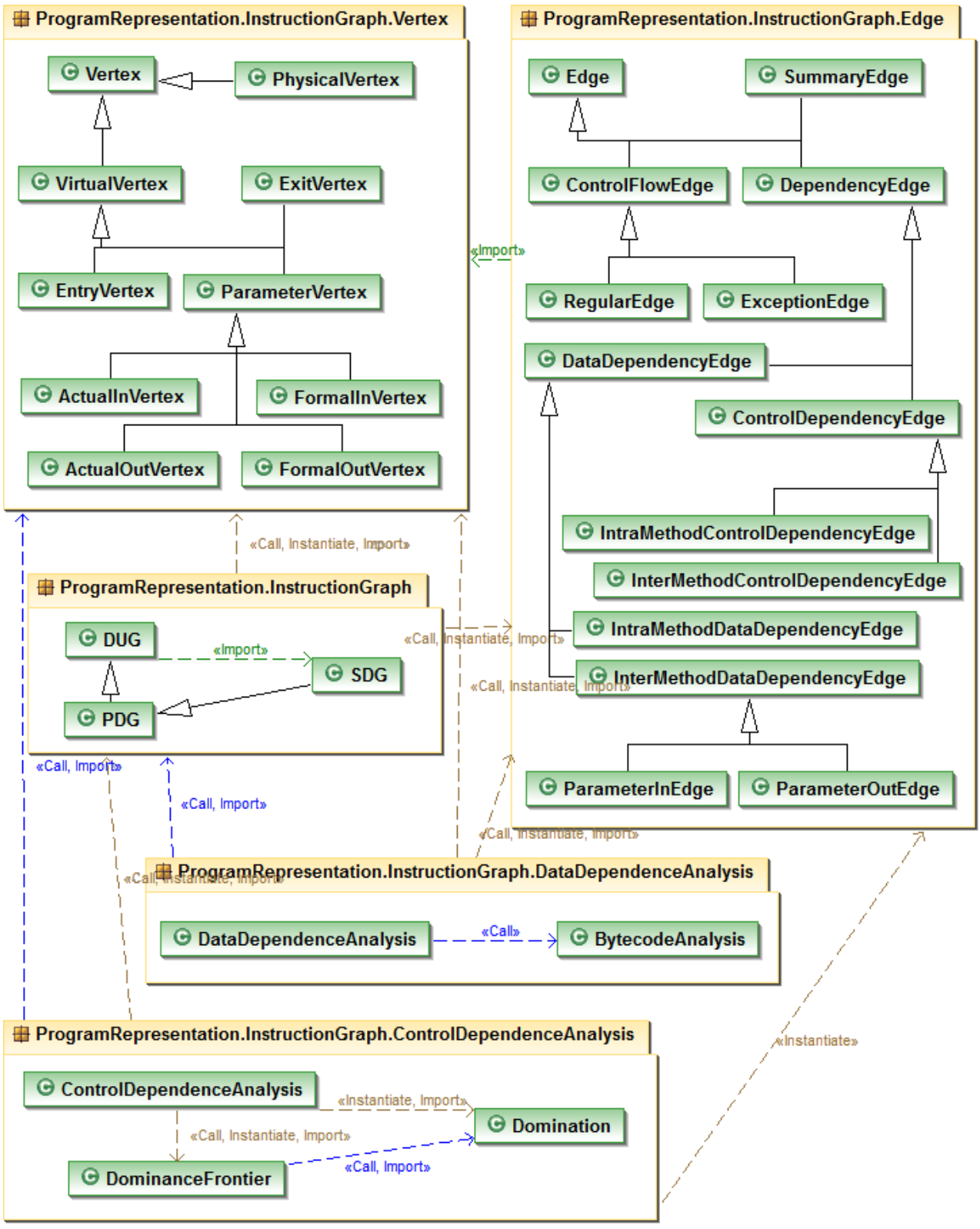


Figura 4.4: Diagrama de Pacotes e de Classes do Pacote `ProgramRepresentation.InstructionGraph` e seus descendentes.

- *Um desvio múltiplo (switch)*: o bloco atual encerra e se conecta a todos os blocos cujos labels são os alvos do desvio;
- *Um retorno (return)*: o bloco atual encerra e não se conecta ao próximo. Ao invés disso, ele se conecta com o vértice de saída do grafo;

- *Um lançamento de exceção (athrow)*: o bloco atual encerra, e se o bloco atual pertence a um bloco `try-catch` que pode tratar a exceção, nada acontece; caso contrário o vértice é conectado com o vértice de saída do grafo.

Quando o alvo de um desvio é um bloco que ainda não foi lido, uma conexão entre os dois blocos é registrada; no momento da inserção de cada bloco, são inseridas as arestas que correspondem a desvios ainda não resolvidos. No início da leitura do código são registrados todos os blocos `try-catch`, com bloco inicial, bloco final, bloco tratador e exceção tratada. Sempre que um bloco encerra, se ele pertence a um `try-catch` uma aresta de exceção é inserida partindo do novo vértice e chegando ao vértice do bloco tratador. As informações de fluxo de dados são inferidas a partir de determinadas situações, que representam definições e/ou usos de variáveis, e inseridas no bloco atual:

- *Acesso a um atributo de classe (getfield e getstatic)*: representa um uso do atributo acessado;
- *Armazenamento do topo da pilha em um atributo (putfield e putstatic)*: representa uma definição do atributo que recebeu o valor do topo da pilha;
- *Incremento (iinc)*: representa um uso e uma definição da variável incrementada;
- *Empilhamento (load)*: representa um uso da variável empilhada;
- *Armazenamento do topo da pilha (store)*: representa uma definição da variável que recebeu o valor do topo da pilha.

Após a leitura do bytecode, a representação de classes e de casos de teste de acordo com o modelo de entidades, e a geração de *DUG* para cada método, os *DUG*s são aumentados com arestas de dependência de controle e de dados. Além disso, são construídos uma tabela de chamadas de métodos e grafos baseados nessas chamadas. Tais atividades são explicadas a seguir.

4.2.2.1 Pós-Processamento – Análise de Dependência

A criação de fatias de programa implica na computação de dependências entre instruções de programa, para permitir que seja determinado se uma instrução em específico (critério de fatia) afeta (*forward*) ou é afetada por (*backward*) outras instruções. A *análise de dependência* é o processo responsável por determinar as dependências entre instruções de programa pela análise de fluxo de controle e de dados. Uma representação de instruções e de dependências entre elas para o cálculo de fatias é *Grafo de Dependência*, sendo que as instruções são representadas por vértices e as dependências são representadas por arestas. Na implementação da análise de dependência da $SoftVis_{4CA}$ são consideradas a *dependência de controle* e a *dependência de fluxo*, que é a dependência de dados relevante para fatiamento de programa.

DEFINIÇÃO 1. Um vértice j é *dependente de controle* de um vértice i se existe um caminho P a partir de i para j tal que j pós-domina todos os vértices em P, exceto i e j, e i não é pós-dominado

por j – um vértice i é pós-dominado por um vértice j se todos os caminhos a partir de i até o vértice de saída do grafo passa por j.

DEFINIÇÃO 2. Um vértice j é *dependente de fluxo* de um vértice i se o valor de uma variável x é definido em i e é usado em j, sendo que existe um caminho de i para j sem que o valor de x seja redefinido (a definição de x no vértice i é uma definição alcançável para o vértice j).

O *DUG* de cada método é submetido à uma análise de dependência intra-método para aumentá-los com arestas de dependência de controle e de dados. O resultado final da análise consiste em um *Grafo de Dependência de Programa* (do inglês *Program Dependence Graph – PDG*) (Kuck et al., 1981, Ottenstein e Ottenstein, 1984, Ferrante et al., 1987) para cada método, representado por instância da classe `PDG`, do pacote `ProgramRepresentation.InstructionGraph`. Além disso, a partir de um método inicial informado pelo usuário, métodos são combinados pela análise de dependência de controle inter-métodos, gerando um objeto da classe `SDG`, do pacote `ProgramRepresentation.InstructionGraph`. O grafo combinado é submetido à uma análise de dependência de dados inter-métodos, resultando em um *Grafo de Dependência de Sistema* (do inglês *System Dependence Graph – SDG*) (Horwitz et al., 1988).

A classe `ControlDependenceAnalysis`, do pacote `ProgramRepresentation.InstructionGraph.ControlDependenceAnalysis`, é responsável pela análise de dependência de controle e criação de arestas para representar as ocorrências de dependência de controle. Quando um *DUG* é submetido à análise, é enviado para tal classe, também, o tipo de análise, que pode ser intra-método ou inter-métodos. No caso da análise intra-método, a classe `ControlDependenceAnalysis` utiliza as classes `Domination`, que calcula os pós- dominadores dos vértices do grafo, e `DominanceFrontier`, que calcula as fronteiras de dominância do grafo[4]. De acordo com as fronteiras de dominância, a classe `ControlDependenceAnalysis` cria as arestas de dependência de controle intra-método (instâncias da classe `IntraMethodControlDependencyEdge`, do pacote `ProgramRepresentation.InstructionGraph.Edge`).

Diferentemente, a análise de dependência de controle inter-métodos combina grafos de diferentes métodos por chamadas de métodos a partir de um método em específico (método inicial), gerando um *PDG* combinado. A classe `ControlDependenceAnalysis` leva em consideração o tipo de combinação entre *PDG*s de métodos a ser realizada, que pode ser *forward* ou *backward*, e o método inicial para a combinação. No caso da combinação *forward*, as instruções dos vértices do grafo do método inicial são percorridas à procura de chamadas de métodos. Quando uma chamada é encontrada, o grafo do método chamado é adicionado ao grafo combinado e o vértice que contém a chamada é conectado com o vértice de entrada do grafo chamado (são criadas arestas de fluxo de controle e de dependência de controle inter-métodos, que são instâncias das classes `RegularEdge` e `InterMethodControlDependencyEdge`, do pacote `ProgramRepresentation.InstructionGraph.Edge`). Esse processo é realizado recursivamente, isto é, o mesmo processa-

[4]As classes `Domination` e `DominanceFrontier`, bem como as classes auxiliares que estas utilizam (que estão no pacote `ProgramRepresentation.Graph.ControlDependenceAnalysis`) foram reutilizadas de Scale Compiler Group (2005).

mento realizado em relação ao grafo do método inicial é realizado também em relação aos grafos secundários adicionados no grafo combinado. A combinação *backward* é realizada de maneira semelhante, sendo a única diferença que as instruções dos vértices dos grafos de todos os métodos são percorridos em busca de chamadas para o método inicial (processo recursivo também).

A classe `DataDependenceAnalysis`, do pacote `ProgramRepresentation.InstructionGraph.DataDependenceAnalysis`, é responsável pela análise de dependência de dados e criação de arestas para representar as ocorrências de dependência de dados. Quando um *DUG* é submetido à análise, é enviado para tal classe, também, o tipo de análise, que pode ser intra-método ou inter-métodos. No caso da análise intra-método, os vértices do grafo são percorridos em busca de dois vértices, um que define e outro que usa a mesma variável. Quando encontrados, é verificado se o vértice que usa a variável é alcançável pelo vértice que define, sem redefinições no meio do caminho de um para o outro. No caso de ser verdadeiro, uma aresta de dependência de dados é criada entre os dois vértices, a qual é uma instância da classe `IntraMethodDataDependencyEdge`, do pacote `ProgramRepresentation.InstructionGraph.Edge`.

No caso da análise ser inter-métodos, o grafo combinado é aumentado com nós e arestas que representam a passagem de parâmetros – os nós são chamados de *actual-in*, *actual-out*, *formal-in* e *formal-out* (representados pelas classes `ActualInVertex`, `ActualOutVertex`, `FormalInVertex` e `FormalOutVertex`, do pacote `ProgramRepresentation.InstructionGraph.Vertex`), que são conectados por arestas *parameter-in* e *parameter-out* (representadas pelas classes `ParameterInEdge` e `ParameterOutEdge`, do pacote `ProgramRepresentation.InstructionGraph.Edge`). As instruções bytecode de vértices do grafo combinado são percorridas em busca de chamadas de métodos. Quando encontrado um vértice v que chama um vértice $entry$, são criados:

- Vértices *actual-in* para cada argumento passado por parâmetro por v, bem como arestas de dependência de controle conectando o vértice v com os vértices *actual-in*;
- Vértices *formal-in* para cada parâmetro recebido por $entry$, bem como arestas de dependência de controle conectando o vértice $entry$ com os vértices *formal-in*;
- Arestas *parameter-in* conectando cada *actual-in* com o seu respectivo *formal-in*.

Se o método chamado retorna algum valor (retorno não sendo `void`), são criados:

- Um vértice *actual-out* para receber o retorno do método chamado, bem como uma aresta de dependência de controle conectando o vértice v com o vértice *actual-out*;
- Um vértice *formal-out* para o retorno do método chamado, bem como uma aresta de dependência de controle conectando o vértice $entry$ com o vértice *formal-out*;
- Uma aresta de dependência de dados conectando o vértice $exit$ do método chamado ao vértice *formal-out*;
- Uma aresta *parameter-out* conectando o vértice *formal-out* com o vértice *actual-out*.

Após a inserção dos vértices e das arestas para passagem de parâmetros, da mesma maneira que na análise intra-método, os vértices do grafo são percorridos em busca de dois vértices, um que define e outro que usa a mesma variável. Quando encontrados, é verificado se o vértice que usa a variável é alcançável pelo vértice que define, sem redefinições no meio do caminho de um para o outro. No caso de ser verdadeiro, uma aresta de dependência de dados é criada em uma das seguintes maneiras:

- Se os vértices $v1$ que usa e $v2$ que define a variável em questão são vértices que representam instruções, e $v1$ possui um vértice *actual-in* que usa a variável de um vértice *formal-out* de $v2$, é inserida uma aresta do *formal-out* para o *actual-in*;
- Se os vértices $v1$ que usa e $v2$ que define a variável em questão são vértices que representam instruções, e $v1$ possui um vértice *actual-in* que usa uma variável definida em $v2$, é inserida uma aresta de $v2$ para o *actual-in*;
- Se os vértices $v1$ que usa e $v2$ que define a variável em questão são vértices que representam instruções, e $v1$ usa a variável de um vértice *formal-out* de $v2$, é inserida uma aresta do *formal-out* para $v1$;
- Se o vértice $v2$ que define a variável em questão é um vértice de entrada de método, é inserida uma aresta do vértice *formal-in* (do vértice $v2$) que representa a variável para $v1$.

4.2.3 Tabela de Chamadas de Métodos

Um fato importante de programa são as chamadas entre métodos. Na ferramenta $SoftVis_{4CA}$ é criada uma estrutura de dados com base nas chamadas entre métodos – estrutura utilizada para diversos propósitos, como para a criação de grafos de chamadas em nível de método e em nível de classe e para mapeamento de cores baseado na métrica *fan-in*. O diagrama de classes que implementam a tabela de chamadas de métodos é ilustrado na Figura 4.5.

A classe `CallTableBuilder`, do pacote `ProgramRepresentation.CallTable`, é responsável por criar a estrutura de dados com base nas chamadas de métodos, representada pela classe `CallTable`, que contém três tabelas – em nível de método (atributo `methodTable`), classe (atributo `classTable`) e pacote (atributo `packageTable`). A partir de um *projeto* criado na fase inicial da *Análise Estática*, o método `buildCallTables()`, da classe `CallTableBuilder`, percorre todos os grafos de *métodos* (*DUG*s) das *classes* e dos *casos de teste* do *projeto* em busca de vértices que contém chamadas para outros métodos. Quando um vértice de *DUG* de um método possui uma instrução bytecode de chamada para método, tal informação é processada em nível de método, classe e pacote, como segue:

- Na tabela de métodos é relacionado o método chamador com o método chamado, guardando a quantidade de chamadas entre eles – quando é inserida tal relação, a quantidade é iniciada em 1, e quando a relação é existente, a quantidade é incrementada;
- Na tabela de classes é relacionada a classe do método chamador com a classe do método chamado, guardando o método chamado e a quantidade de chamadas (similar ao que é feito na tabela de métodos);

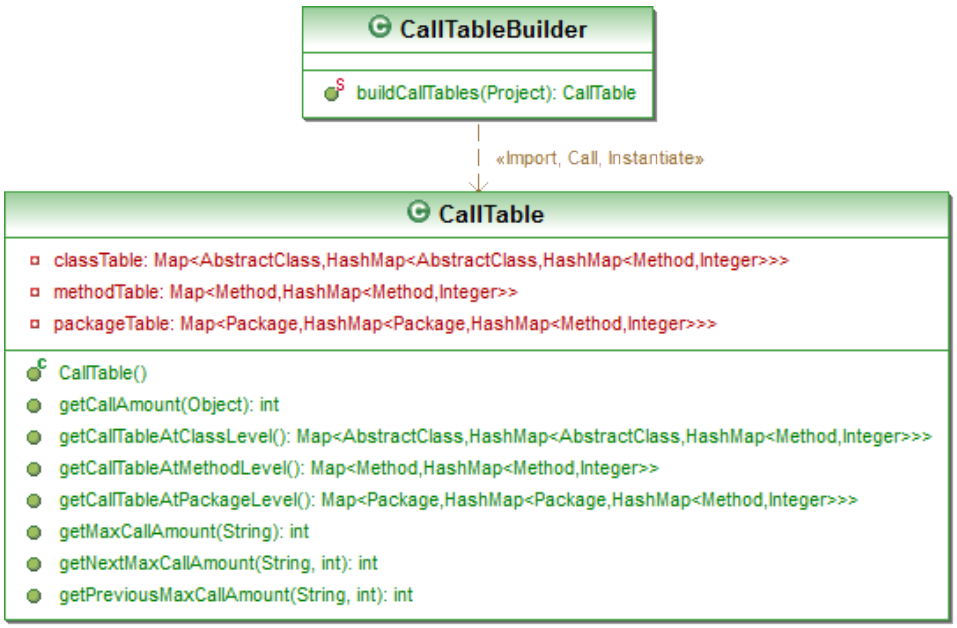


Figura 4.5: Diagrama de Classes do Pacote `ProgramRepresentation.CallTable`.

- Na tabela de pacote é relacionado o pacote da classe do método chamador com o pacote da classe do método chamado, guardando o método chamado e a quantidade de chamadas (similar ao que é feito na tabela de métodos).

Após a construção das tabelas de chamadas, o objeto `CallTable` criado é guardado no objeto *projeto* que foi utilizado para a construção das tabelas.

4.2.4 Grafos baseados em Chamadas de Métodos

As *Visões Inter-Classes* e *Inter-Métodos* da $SoftVis_{4CA}$ projetam grafos baseados em chamadas de métodos, em nível de classe e de método. Sendo assim, a geração dessas visões implica na construção de estruturas de dados (grafos) previamente. O pacote `ProgramRepresentation.ClassGraph` contém toda implementação de grafo de classes baseado em chamadas de métodos entre elas (o diagrama de classes do pacote é ilustrado na Figura 4.6). A partir de um *projeto*, a classe `ClassGraphBuilder` obtém e percorre a tabela de chamadas em nível de classe, inserindo em um objeto instanciado da classe `ClassGraph` as classes, que são representadas por vértices do grafo, e as chamadas entre elas, que são representadas por arestas do grafo. As arestas são instâncias da classe `CallEdge`, que guardam os métodos chamados e a quantidade de chamadas de cada método entre duas classes.

De maneira similiar, o pacote `ProgramRepresentation.MethodGraph` contém a implementação de grafo de chamada entre métodos (o diagrama de classes do pacote é ilustrado na Figura 4.7). A partir de um *projeto*, a classe `MethodGraphBuilder` obtém e percorre a tabela de chamadas em nível de método, inserindo em um objeto instanciado da classe `MethodGraph` os métodos, que são representados por vértices do grafo, e as chamadas entre eles, que são representadas por arestas do grafo. As arestas são instâncias da classe `CallEdge`, que guardam a quantidade de chamadas entre dois métodos.

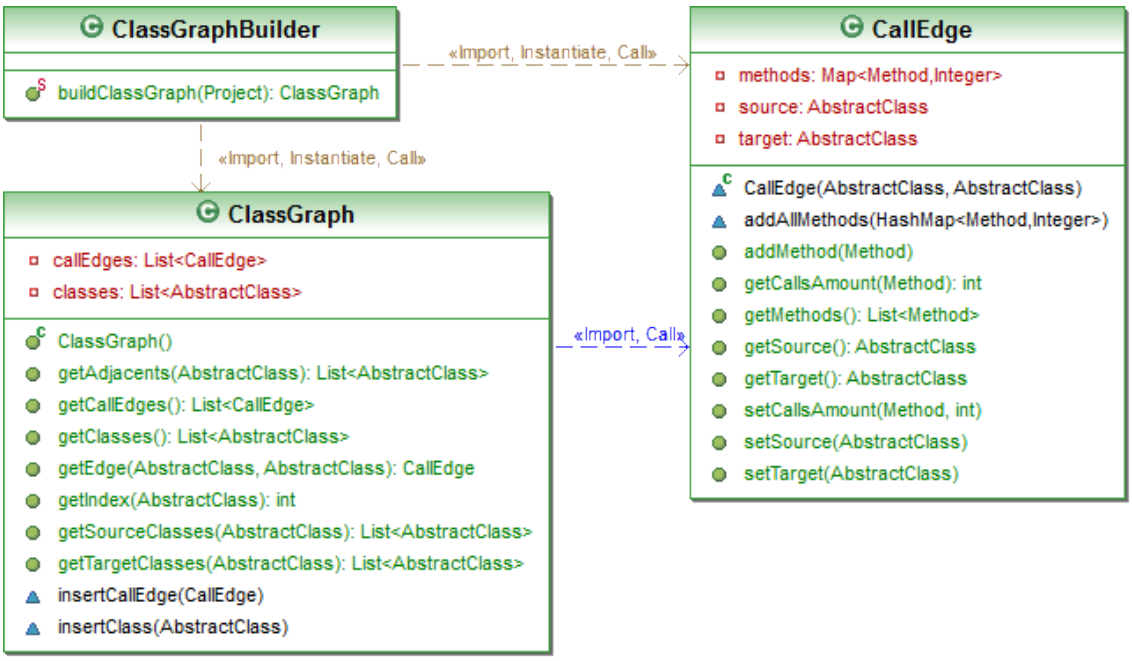


Figura 4.6: Diagrama de Classes do Pacote `ProgramRepresentation.ClassGraph`.

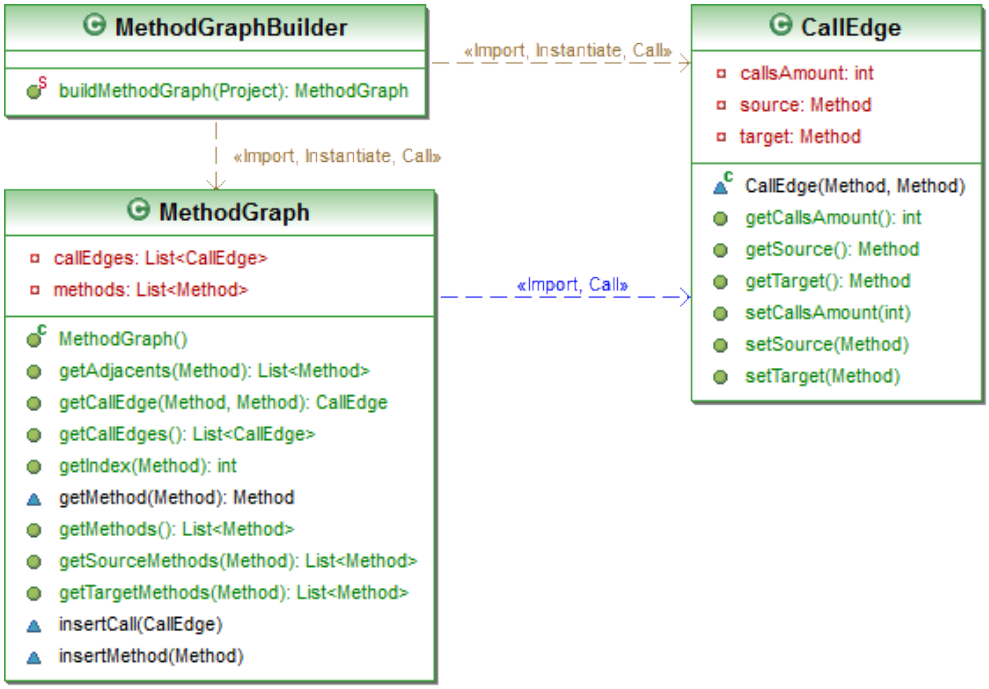


Figura 4.7: Diagrama de Classes do Pacote `ProgramRepresentation.MethodGraph`.

4.2.5 Preparação para a Análise Dinâmica

Para que a análise dinâmica de um programa aconteça, é necessário que o programa em tempo de execução forneça as informações necessárias sobre as instruções executadas para a ferramenta (monitoramento do teste), bem como o código do programa seja carregado na máquina virtual, para que sua execução seja possível.

O processo de instrumentação consiste na inserção de chamadas de métodos estáticos da classe `DynamicAnalysis` (do pacote `CodeAnalysis.Dynamic`) em determinados pontos do código

de métodos, que enviam informações sobre os locais executados para a *Análise Dinâmica* fazer o processamento necessário. A instrumentação é realizada pela classe `Instrumentation`, do pacote `CodeAnalysis.Static` (vide Figura 4.2), que intercepta o processo de leitura do código da unidade inserindo as novas instruções conforme necessário. Os pontos de inserção são:

- *Início do código de um método*: uma chamada ao método `UNIT` da classe `DynamicAnalysis` é inserida no início do código de métodos, enviando como parâmetros o nome, o descritor e o nome da classe do método;
- *Label*: uma chamada ao método `LABEL` da classe `DynamicAnalysis` é inserida após cada label do código, enviando como parâmetro o label;
- *Retorno*: uma chamada ao método `RETURN` da classe `DynamicAnalysis` é inserida antes de cada retorno de método, comum ou de exceção, sem enviar qualquer informação.

Após as classes e os casos de testes terem sido lidos e as classes terem passado pelo processo de instrumentação, os mesmos são inseridos no projeto a qual pertencem, e então são carregados na máquina virtual. Os métodos responsáveis pelo carregamento na máquina virtual são os métodos `load(Class clazz)` e `load(TestCase testCase)` da classe `MyClassLoader`, do pacote `CodeAnalysis.Static` (vide Figura 4.2).

4.3 Análise Dinâmica

A criação de fatias dinâmicas implica na coleta de informações em tempo de execução de um programa, sendo necessário monitorar a execução de casos de teste. A análise dinâmica implementada na $SoftVis_{4CA}$ registra as instruções executadas por casos de testes criados usando *JUnit*, construindo um grafo de fluxo de controle do caminho executado (o diagrama de pacotes e de classes responsáveis pela análise dinâmica e que representam testes é ilustrado na Figura 4.8).

É possível realizar dois tipos de teste estrutural, sendo eles teste de unidade e teste de módulo. O teste de unidade define uma única unidade (método) a ser analisada durante a execução do caso de teste. O teste de módulo define uma unidade principal e um conjunto de unidades secundárias a serem analisadas durante a execução, sendo as unidades secundárias métodos chamados pela unidade principal.

A classe abstrata `StructuralTesting`, do pacote `Test`, representa um teste estrutural, e define a base de todos atributos e métodos necessários para a criação e execução de um teste estrutural. As sub-classes `IntraMethodTest` e `InterMethodTest` definem o escopo do teste (unidade ou módulo). Um teste é criado a partir de um caso de teste e de uma unidade principal. Quando o teste é de unidade, o grafo de fluxo do teste é o grafo de fluxo da unidade principal, e quando o teste é de módulo, o grafo de fluxo do teste é a combinação do grafo da unidade principal com os grafos das unidades secundárias. A execução de um teste envolve as seguintes operações: (i) preparação da análise dinâmica, que envolve descartar todas as informações de um teste executado anteriormente, bem como registrar as unidades do teste a serem monitoradas, por meio de um grafo; (ii) execução do código do caso de teste pela máquina virtual Java; (iii) processamento do resultado, por meio da

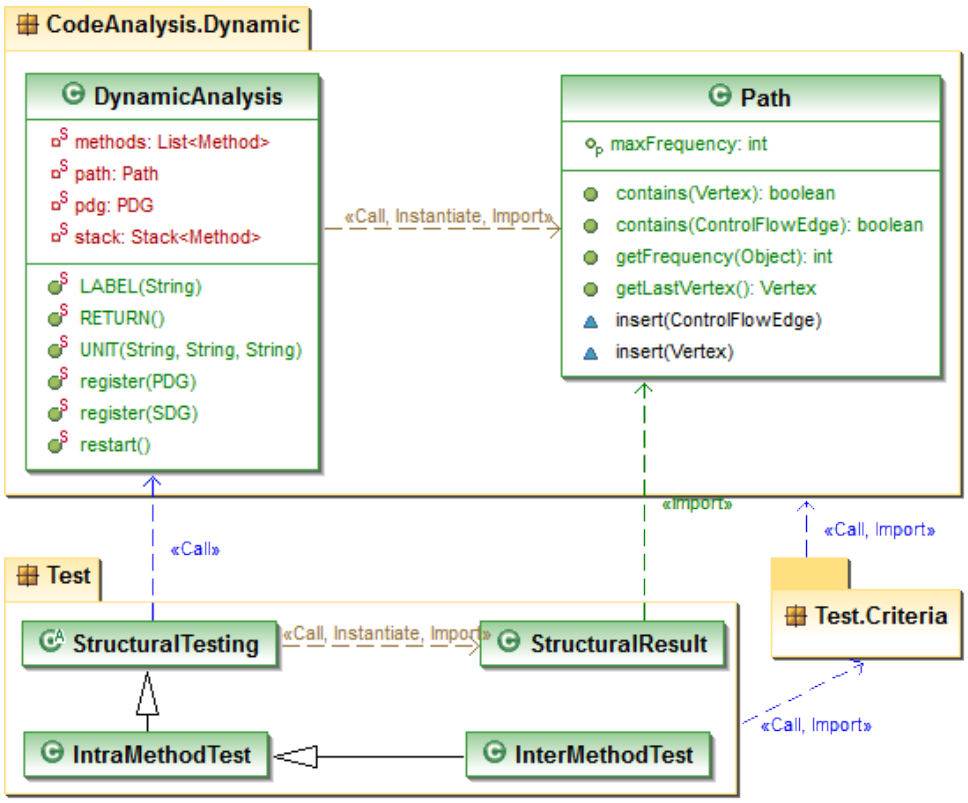


Figura 4.8: Diagrama de Pacotes e de Classes dos Pacotes `CodeAnalysis.Dynamic`, `Test` e `Test.Criteria`.

verificação dos critérios de abrangência previamente definidos (os critérios são implementados no pacote `Test.Criteria`).

A classe `DynamicAnalysis`, do pacote `CodeAnalysis.DynamicAnalysis`, é responsável pelo processamento de informações enviadas pela instrumentação inserida no bytecode pela análise estática. Os métodos `LABEL`, `UNIT` e `RETURN` são chamados pela execução do teste e processam as informações obtidas a partir da execução do teste. A classe `Path` representa o caminho registrado pela classe `DynamicAnalysis`, em que tal caminho é composto pelos vértices e pelas arestas visitados durante a execução e a frequência de visitas de cada um.

Uma chamada ao método `UNIT` indica à análise dinâmica que uma nova unidade de teste tomou o controle da execução. Como diversas unidades podem estar ativas ao mesmo tempo, esperando o retorno de outra unidade, a análise dinâmica utiliza uma pilha de unidades para simular a pilha de execução da máquina virtual Java. A unidade no topo da pilha é considerada a unidade ativa no momento. Com as informações recebidas nos parâmetros do método `UNIT` (classe, nome e descritor da unidade) é possível determinar se a unidade faz parte ou não do teste. Se sim, a unidade é empilhada; caso contrário ela é ignorada e um objeto nulo é empilhado, indicando que todas as outras chamadas ao método `LABEL` devem ser ignorados até que esta unidade retorne. Quando o método `LABEL` é chamado, significa que um label foi encontrado no código, podendo ou não ser o início de um bloco de código e um vértice do grafo da unidade ativa. Inicialmente o topo da pilha é verificado; se for nulo, a chamada é desconsiderada. Se o grafo da unidade ativa possui um vértice que inicia com o label recebido como parâmetro da chamada, o vértice é adicionado ao caminho; caso contrário o label é ignorado pois não representa o início de um novo bloco de código. Quando um vértice é inserido no

caminho, também é inserida a aresta que o conecta ao último vértice inserido anteriormente (exceto o vértice de entrada). Uma chamada ao método `RETURN` significa que a unidade ativa retornou, sendo tal desempilhada, e a unidade anterior passa a ser a unidade ativa. O mesmo acontece com unidades que não fazem parte do teste, seus objetos nulos são desempilhados liberando a execução para a unidade anterior (Martins, 2007).

O caminho percorrido pela execução do caso de teste e os resultados dos critérios de abrangência definidos para o teste compõem o seu resultado estrutural. Essas informações são armazenadas na classe `StructuralResult`, derivada da classe `Result` do *framework JUnit*, que armazena o resultado funcional do teste.

4.4 Representações Visuais

A *Camada de Visualização* da $SoftVis_{4CA}$ provê seis visões. A *Visão Inter-Classes* apresenta as chamadas de métodos entre classes usando projeção hiperbólica (descrita na Seção 3.1); a *Visão Inter-Métodos* apresenta as chamadas entre métodos usando, também, uma projeção hiperbólica; a *Visão Inter-Instruções* mostra as dependências entre instruções de programa agrupadas pelos seus métodos, usando uma projeção de agregados; a *Visão Estrutural* apresenta a organização de programa em uma estrutura hierárquica usando a técnica *Treemap* (descrita na Seção 3.1); a *Visão de Distribuição de Instruções* apresenta a distribuição de conjuntos de instruções em classes usando a técnica *Barras e Listras* (similar ao *plug-in Visualiser* descrito na Seção 3.4.1); e a *Visão de Bytecode* exibe as instruções bytecode de programa.

4.4.1 *Visão Inter-Classes* e *Visão Inter-Métodos*

Diversas técnicas propostas para a mineração de aspectos são realizadas em nível de métodos, levando em consideração as chamadas entre eles (Breu e Krinke, 2004, Krinke e Breu, 2005, Abait et al., 2008, Tonella e Ceccato, 2004, Marin et al., 2007). A $SoftVis_{4CA}$ provê duas visões para a apresentação de chamadas de métodos usando projeção hiperbólica – uma em nível de classe (*Visão Inter-Classes*) e outra em nível de método (*Visão Inter-Métodos*) – em que as classes e os métodos de um programa são mapeados em nós, e as chamadas entre tais unidades são mapeadas em arestas. A implementação dessas visões, juntamente com a implementação da projeção em si, pertencem ao pacote `View.Hyperbolic` e seus descendentes (vide Figura 4.9).

Os dados necessários para gerar as duas visões foram criados na fase da análise estática, que são as estruturas `ClassGraph` e `MethodGraph` (dos pacotes `ProgramRepresentation.ClassGraph` e `ProgramRepresentation.MethodGraph`). Essas estruturas são passadas para as classes `ClassGraphToPrefuseGraph` e `MethodGraphToPrefuseGraph`, do pacote `View.Hyperbolic.Data`, que constroem uma estrutura `Graph` do PREFUSE[5] para cada uma das estruturas (`ClassGraph` e `MethodGraph`). Cada objeto `Graph` é guardado em um objeto instanciado da classe `PrefuseGraph`, do pacote `View.Prefuse.Data`, juntamente com o objeto gerado pela análise estática (`ClassGraph` e `MethodGraph`). Então, os objetos `PrefuseGraph` são passados para as classes `InterClassesView` e `InterMethodsView`, do pacote `View.Hyper-`

[5] PREFUSE é um conjunto de ferramentas de software para a criação de visualizações de dados interativas (Prefuse, 2012).

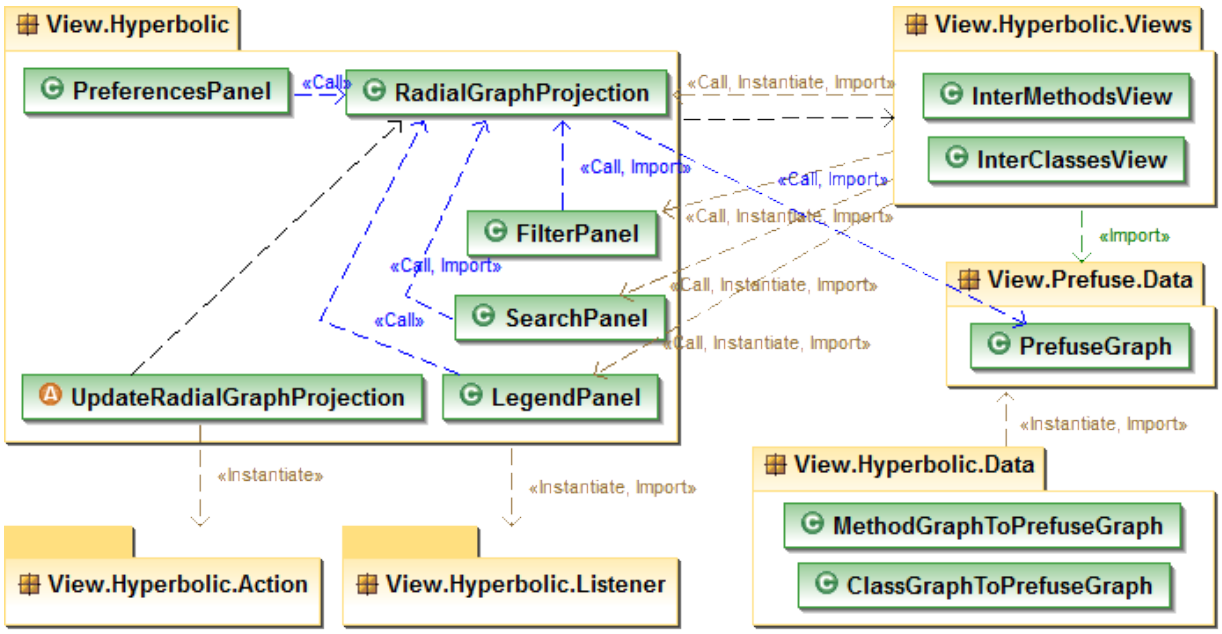


Figura 4.9: Diagrama de Pacotes e de Classes do Pacote `View.Hyperbolic` e seus descendentes.

`bolic.Views`, que estendem a `JInternalFrame` para a apresentação da projeção hiperbólica. A projeção hiperbólica é implementada na classe `RadialGraphProjection`, do pacote `View.Hyperbolic`, que utiliza a classe `RadialTreeLayout` do PREFUSE. Todas as *actions*[6] da projeção são implementadas no pacote `View.Hyperbolic.Action`.

Juntamente com cada visão, são apresentados, também: (i) um filtro de distância (provido pela classe `FilterPanel`), que pode ser utilizado para a visão de uma certa distância de nós conectados a partir de um nó em foco; (ii) uma opção de pesquisa (provida pela classe `SearchPanel`), para o destaque de nós a partir de uma sequência de caracteres como palavra chave (referente ao nome de classe ou de método); (iii) e uma legenda que contém o significado de cada tipo de nó (provida pela classe `LegendPanel`).

Adicionalmente, há preferências (providas pela classe `PreferencesPanel`) descritas na Seção 4.6, para que o usuário possa personalizar a projeção. Quando uma preferência é modificada, a projeção é atualizada pelo aspecto `UpdateRadialGraphProjection`. Os *listeners* dos componentes utilizados para o filtro de distância e para as preferências são implementados no pacote `View.Hyperbolic.Listener`.

4.4.2 *Inter-Instruções*

A *Visão Inter-Instruções* mostra grafos em nível de instruções de programa, em que as instruções são mapeadas em nós e as dependências entre elas são mapeadas em arestas. Cada método do programa tem seu próprio grafo, e grafos podem ser combinados por chamadas de métodos a partir de ou para um método selecionado (*forward* ou *backward*). A implementação dessas visões, juntamente com a implementação da projeção de agregados em si, pertencem ao pacote `View.Aggregate` e seus descendentes (vide Figura 4.10).

[6] *Actions* são responsáveis por realizar uma série de operações em uma visualização, e normalmente processam itens visuais para definir atributos visuais, como cores, por exemplo.

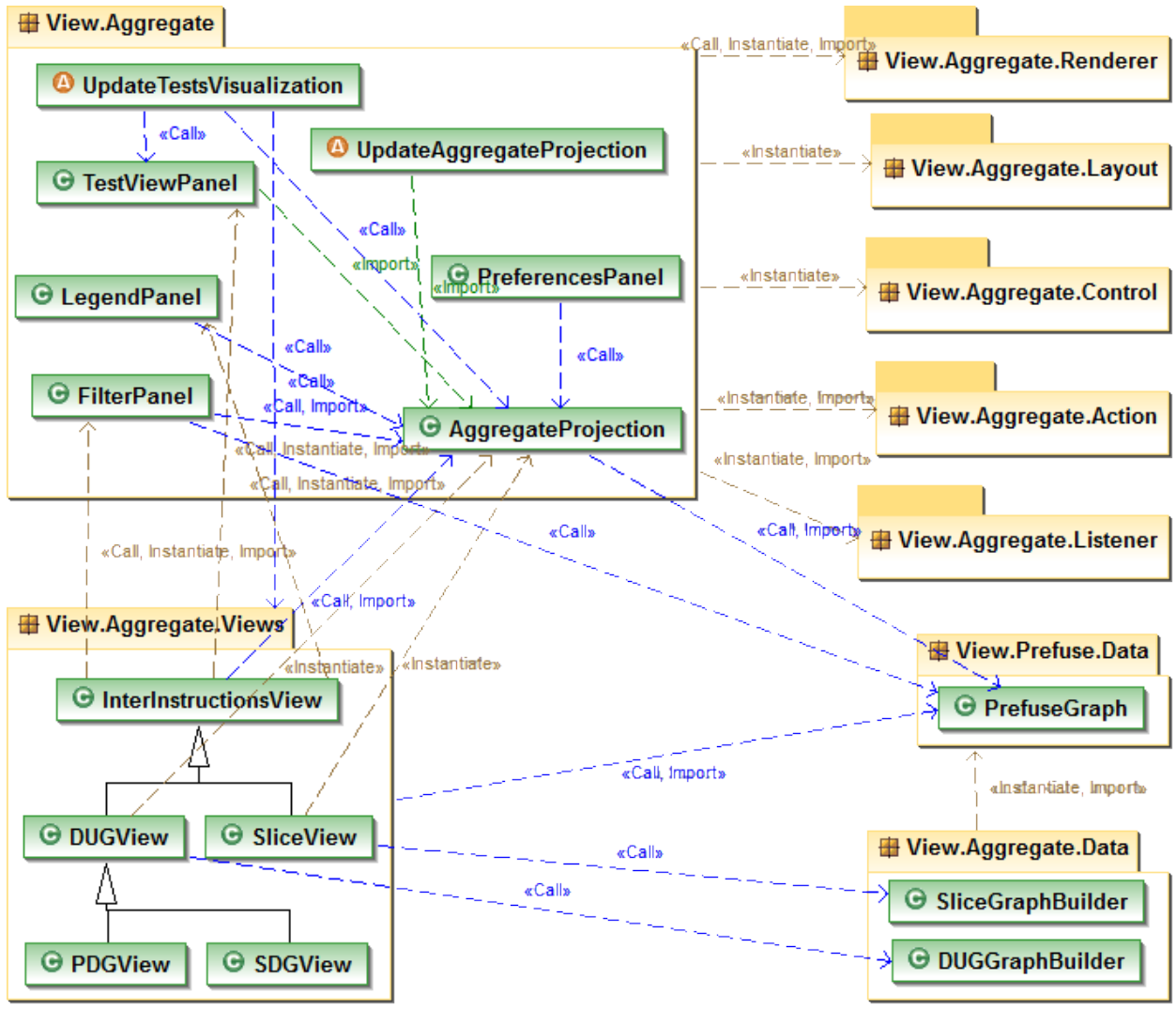


Figura 4.10: Diagrama de Pacotes e de Classes do Pacote `View.Aggregate` e seus descendentes.

Os grafos (de métodos e combinado) foram criados na fase da análise estática, que são as estruturas `PDG` e `SDG` (do pacote `ProgramRepresentation.InstructionGraph`). Essas estruturas são passadas para a classe `InterInstructionsView` ou para alguma de suas extensões, do pacote `View.Aggregate.Views`, que chama a classe `DUGGraphBuilder` (ou `SliceGraphBuilder`, no caso do grafo ser de uma fatia), do pacote `View.Aggregate.Data`, que por sua vez constrói estruturas `Graph` do PREFUSE para cada grafo. Cada objeto `Graph` é guardado em um objeto instanciado da classe `PrefuseGraph`, do pacote `View.Prefuse.Data`, juntamente com o grafo gerado pela análise estática. A classe `InterInstructionsView` e suas extensões estendem a `JInternalFrame` para a apresentação da projeção de agregados. A projeção de agregados é implementada na classe `AggregateProjection`, do pacote `View.Aggregate`, que utiliza a classe `ForceDirectedLayout` do PREFUSE. A classe `AggregateProjection` utiliza, também, as classes dos pacotes `View.Aggregate.Renderer`, `View.Aggregate.Layout` e `View.Aggregate.Control`, e todas as *actions* da projeção são implementadas no pacote `View.Aggregate.Action`.

Juntamente com cada visão, são apresentados, também: (i) um filtro de distância, que pode ser utilizado para a visão de uma certa distância de nós conectados a partir de um nó em foco; (ii) um filtro

de visibilidade de tipos de nós; (iii) um filtro de visibilidade de tipos de arestas; (iv) uma tabela que lista os testes criados e executados, para a visualização de instruções executadas no grafo; (v) e uma legenda que contém o significado de cada tipo de nó e de cada tipo de aresta. Os filtros são providos pela classe `FilterPanel`, a tabela de testes pela classe `TestViewPanel` e a legenda pela classe `LegendPanel`. A tabela de testes é atualizada pelo aspecto `UpdateTestsVisualization`.

Adicionalmente, há preferências (providas pela classe `PreferencesPanel`) descritas na Seção 4.6, para que o usuário possa personalizar a projeção. Quando uma preferência é modificada, a projeção é atualizada pelo aspecto `UpdateAggregateProjection`. Os *listeners* dos componentes utilizados para os filtros e para as preferências são implementados no pacote `View.Aggregate.Listener`.

4.4.3 *Visão Estrutural*

A $SoftVis_{4CA}$ provê uma visão estrutural, usando a técnica de visualização *Treemap*, para apresentar a estrutura de programa. As unidades de programa – pacote, classe, caso de teste e método – são mapeadas em retângulos aninhados. A implementação dessa visão pertence ao pacote `View.Treemap` e seus descendentes (vide Figura 4.11).

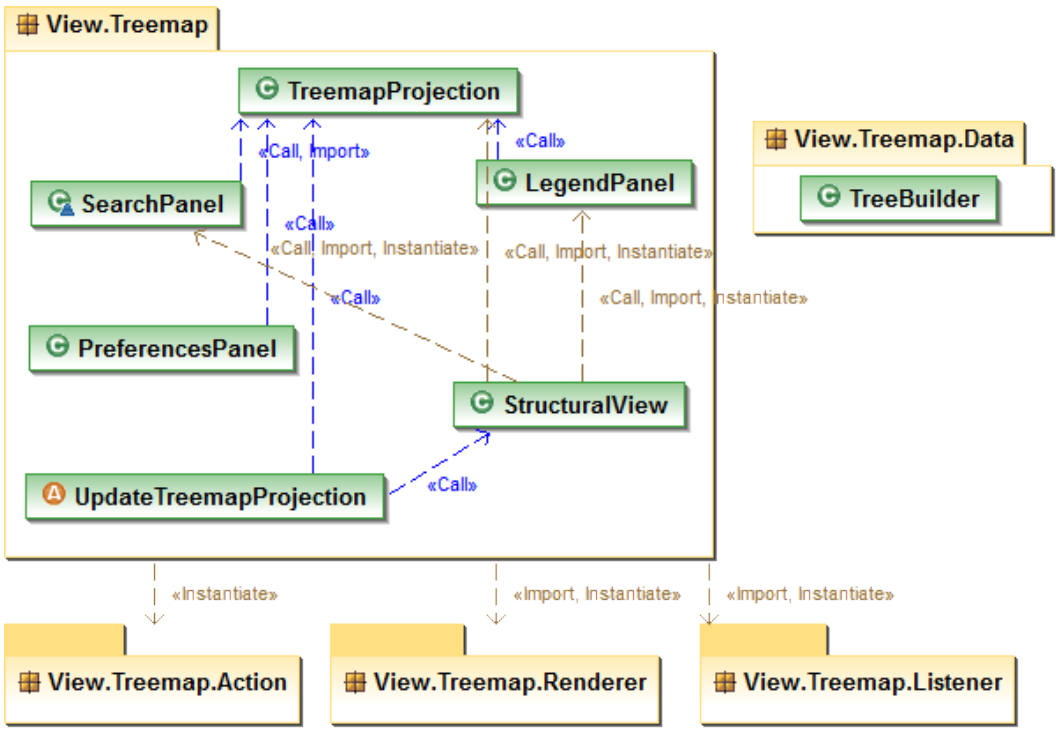


Figura 4.11: Diagrama de Pacotes e de Classes do Pacote `View.Treemap` e seus descendentes.

Os dados necessários para gerar a visão estrutural são organizados pela classe `TreeBuilder`, do pacote `View.Treemap.Data`. A partir de um objeto *projeto*, a classe `TreeBuilder` navega pela composição dos objetos instanciados do modelo de entidades, isto é, percorre todos os pacotes, todas as classes e todos os casos de teste de cada pacote, e todos os métodos de cada classe e de cada caso de teste. Por meio dessa navegação é construída a estrutura `Tree` do PREFUSE. Tal estrutura é passada para a classe `StructuralView`, do pacote `View.Treemap`, que estende a `JInternalFrame` para a apresentação estrutural. A apresentação estrutural é implementada na

classe `TreemapProjection`, do pacote `View.Treemap`, que utiliza a classe `Squarified-TreeMapLayout` provida pelo PREFUSE. A classe `TreemapProjection` utiliza, também, a classe do pacote `View.Treemap.Renderer`, e todas as *actions* da projeção são implementadas no pacote `View.Treemap.Action`.

Juntamente com a visão, são apresentadas, também: (i) uma opção de pesquisa (provida pela classe `SearchPanel`), para o destaque de retângulos a partir de uma sequência de caracteres como palavra chave (referente ao nome da unidade); (ii) e uma legenda que contém o significado de cada tipo de retângulo (provida pela classe `LegendPanel`).

Adicionalmente, há preferências (providas pela classe `PreferencesPanel`) descritas na Seção 4.6, para que o usuário possa personalizar a projeção. Quando uma preferência é modificada, a projeção é atualizada pelo aspecto `UpdateTreemapProjection`. Os *listeners* dos componentes utilizados para as preferências são implementados no pacote `View.Treemap.Listener`.

4.4.4 *Visão de Distribuição de Instruções*

A *Visão de Distribuição de Instruções*, provida pela $SoftVis_{4CA}$ usando a técnica de visualização *Barras e Listras*, apresenta uma visão geral de como conjuntos de instruções são distribuídos em múltiplas classes. As classes são mapeadas em barras e as instruções são mapeadas em listras. A implementação dessa visão pertence ao pacote `View.BarsAndStripes` e seus descendentes (vide Figura 4.12).

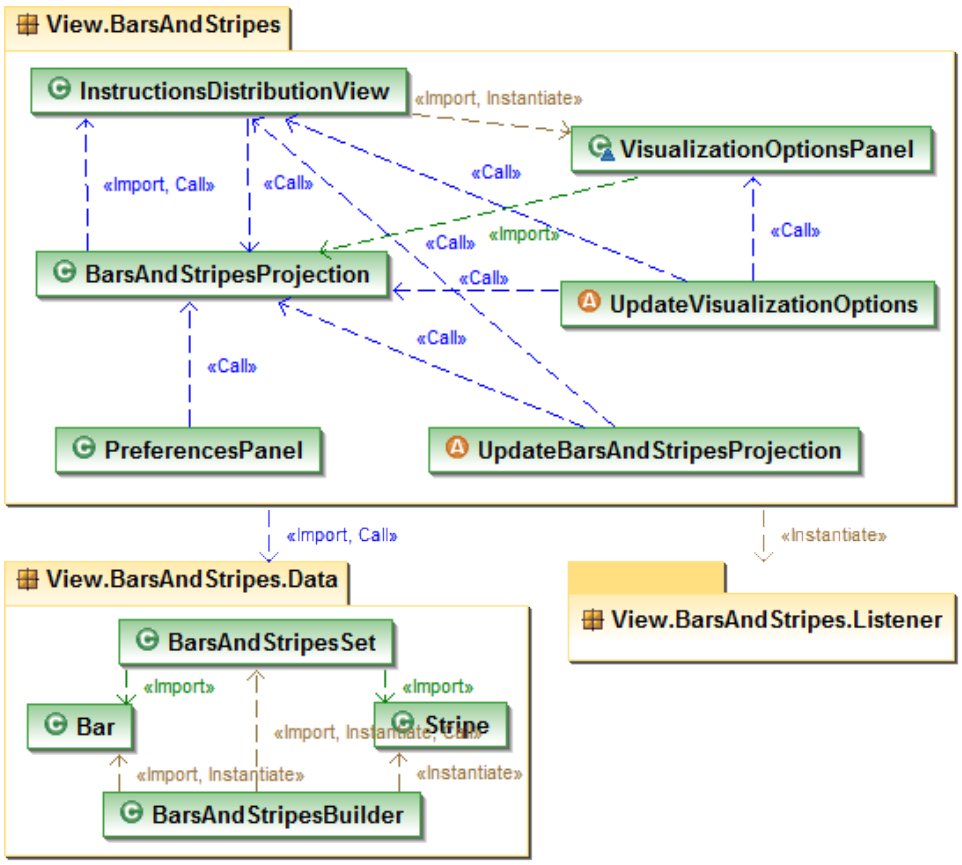


Figura 4.12: Diagrama de Pacotes e de Classes do Pacote `View.BarAndStripes` e seus descendentes.

Os dados necessários para gerar a visão são organizados pela classe `BarsAndStripesBuilder`, do pacote `View.BarsAndStripes.Data`. A partir de um objeto *projeto*, a classe `BarsAndStripesBuilder` percorre as classes e os casos de teste do projeto. Para cada classe e caso de teste, são percorridos os vértices que representam instruções do grafo de cada método. Por meio dessa navegação é construído um conjunto de barras (classes e casos de teste) que contém listras (vértices de instrução), instanciado da classe `BarsAndStripesSet`, do pacote `View.BarsAndStripes.Data`. Tal estrutura é passada para a classe `InstructionsDistributionView`, do pacote `View.BarsAndStripes`, que estende a `JInternalFrame` para a apresentação das barras e listras. A projeção de barras e listras é implementada na classe `BarsAndStripesProjection`, do pacote `View.BarsAndStripes`, na qual recebe o conjunto de barras e listras e configura o tamanho das barras de acordo com a quantidade de instruções bytecode dos vértices representados pelas listras que a barra possui.

Juntamente com a visão, é apresentado, também, um painel de opções para visualização (provido pela classe `VisualizationOptionsPanel`), exibindo em duas tabelas os dois tipos de conjuntos de instruções – fatias criadas e testes executados. Por meio da seleção de um conjunto de instruções em alguma das tabelas, são coloridas as listras que representam instruções que fazem parte da fatia ou que foram executadas por um teste. Cada conjunto de instruções possui uma cor selecionada aleatoriamente, mas elas também podem ser alteradas pela interação do usuário com o botão de cor que cada conjunto possui. O aspecto `UpdateVisualizationOptions` é responsável por atualizar as duas tabelas e a projeção quando uma fatia é criada ou um teste é executado, bem como quando um dos mesmos é removido ou sua cor é alterada.

Adicionalmente, há preferências (providas pela classe `PreferencesPanel`) descritas na Seção 4.6, para que o usuário possa personalizar a projeção. Quando uma preferência é modificada, a projeção é atualizada pelo aspecto `UpdateBarsAndStripesProjection`. Os *listeners* dos componentes utilizados para as preferências são implementados no pacote `View.BarsAndStripes.Listener`.

4.4.5 *Visão de Bytecode*

A *Visão de Bytecode* consiste na exibição de instruções bytecode de programa. A implementação dessa visão pertence ao pacote `View.Bytecode` (vide Figura 4.13). A partir de um objeto *projeto*, a classe `BytecodeView`, que estende a `JInternalFrame` para a listagem de instruções, percorre as classes e os casos de teste, obtendo todos os métodos do projeto. Então, a lista de métodos é passada para as classes `BytecodeOverview` e `BytecodeDetail`, que estendem a *JPanel* e são utilizadas para apresentar uma visão geral e uma visão detalhada das instruções, respectivamente. Os objetos instanciados dessas classes são adicionadas ao objeto instanciado da classe `BytecodeView` para apresentação das instruções.

Por se tratar de listagens, há a existência de barras de rolagem. No pacote `View.Bytecode.Listener` contém os *listeners* que controlam as barras de rolagem da visão geral e da visão de detalhes e garantem a sincronização das duas visões. Em adicional, o aspecto `UpdateBytecode` é responsável por atualizar a visão geral e a visão de detalhes quando a cor de um método é modificada.

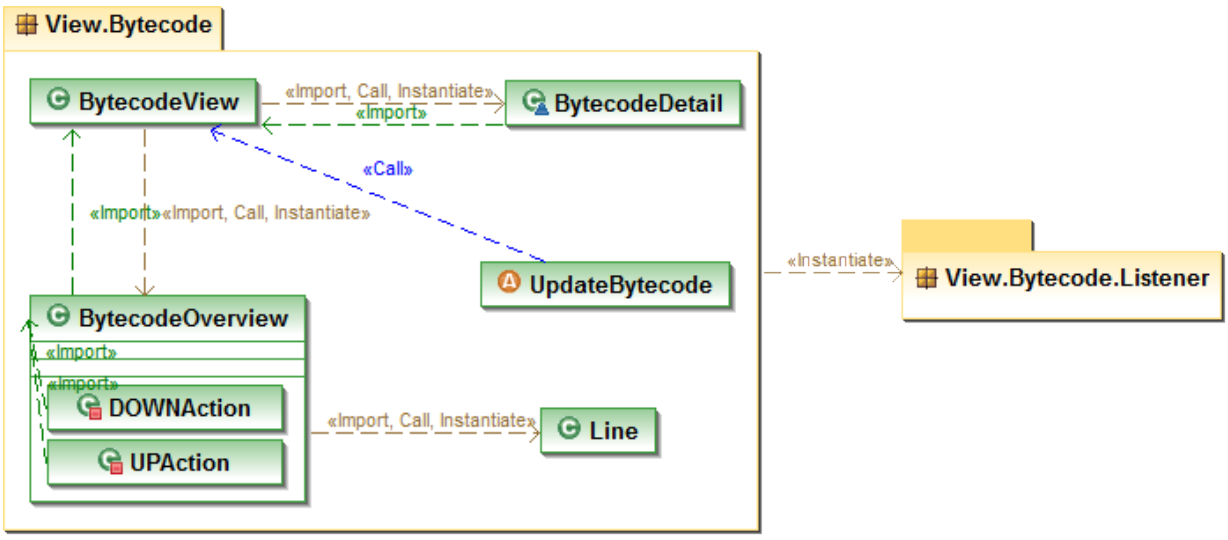


Figura 4.13: Diagrama de Classes do Pacote `View.Bytecode` e seu descendente.

4.4.6 Mapeamento de Cores baseado na Métrica *Fan-in*

Em cada representação visual, as cores dos itens visuais são pré-definidas, e o usuário pode modificar tais cores se desejar (as preferências estão descritas na Seção 4.6). Além disso, há um mapeamento de cores usado nas visões que é baseado na quantidade de chamadas de métodos, uma característica da técnica *Análise de Fan-in* proposta para minerar aspectos (descrita na Seção 2.3.1). A implementação de tal mapeamento pertence ao pacote `View.ColorMappingBasedOnCalls` e seus descendentes (vide Figura 4.14).

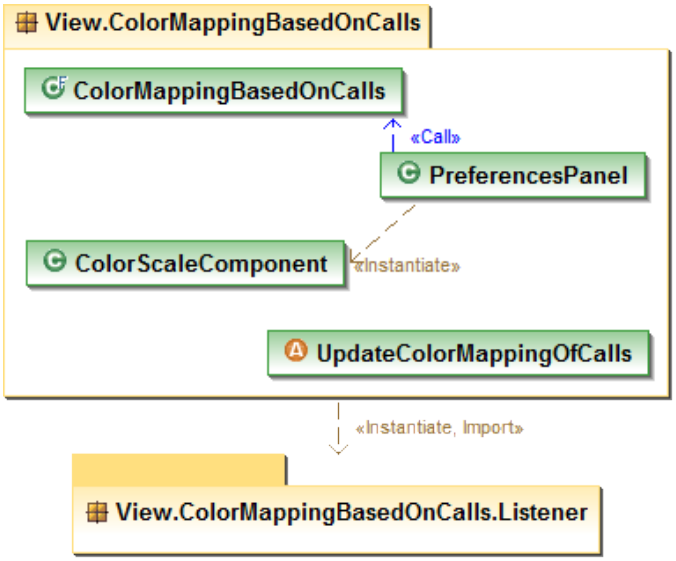


Figura 4.14: Diagrama de Pacotes e de Classes do Pacote `View.ColorMappingBasedOnCalls` e seu descendente.

Os dados necessários para que o mapeamento seja realizado foram criados na fase da *Análise Estática*, que é a estrutura `CallTable`, contendo três tabelas, uma para cada unidade (i.e., pacotes, classes e métodos), que contém a relação de unidades chamadas e chamadoras juntamente com os valores da métrica *fan-in*. A partir de um *projeto*, a classe `ColorMappingBasedOnCalls`, que

estende a `JSlider`, exibe um gradiente de cor do cinza claro para o vermelho, usado para representar os valores obtidos (vermelho representa o valor maior). Visto que a métrica *fan-in* foi calculada em três níveis, o usuário pode escolher qual o tipo de unidade para o mapeamento, sendo padrão a escala de cores em nível de método (neste caso, é usada a tabela de métodos da `CallTable`).

O ponteiro do componente *JSlider* é iniciado no valor máximo de chamadas para o tipo de unidade ativo. Sendo assim, quando a escala de cores é iniciada, é mapeada a cor vermelha saturada (máxima) para os itens visuais que são os máximos chamados. No entanto, é possível filtrar unidades de código com base em valor de *fan-in* por meio da definição de um limiar, permitindo observar as unidades correspondentes nas projeções e, assim, é possível visualizar as unidades com alta frequência de chamadas.

Adicionalmente, há preferências (providas pela classe `PreferencesPanel`, que inclusive utiliza a classe `ColorScaleComponent`) descritas na Seção 4.6, para que o usuário possa personalizar o mapeamento. Os *listeners* dos componentes utilizados para as preferências são implementados no pacote `View.ColorMappingBasedOnCalls.Listener`, e o aspecto `UpdateColorMappingOfCalls` é responsável por atualizar a escala de cores de acordo com um *projeto*.

4.4.7 Fatiamento de Programa

Fatias podem ser criadas e visualizadas como subgrafos, por meio da interação do usuário. Para isso, o usuário escolhe um nó que representa uma instrução em específico (critério de fatia) e o tipo de fatia, que pode ser estática ou dinâmica, a partir de ou para o nó selecionado (fatia *backward* ou *forward*). Os grafos de métodos são utilizados para criar fatias intra-método, e grafos combinados são utilizados para criar fatias inter-métodos. Assim, uma *Visão Inter-Instruções* é criada para apresentar a fatia.

Em abordagens de fatiamento baseadas em grafo de dependência, o critério de fatiamento é identificado como um vértice v no grafo. Na terminologia de Weiser (1984), isso corresponde a um critério (n, V) em que n é o nó no grafo correspondente a v, e V o conjunto de todas as variáveis definidas (no caso de fatiamento *forward*) ou utilizadas (no caso de fatiamento *backward*) em v (Tip, 1995). O diagrama de classes que contém a implementação de fatiamento de programa é ilustrado na Figura 4.15.

4.4.7.1 Fatiamento *Backward*

O fatiamento *backward* intra-método recebe como entrada um *PDG* e um vértice de critério v. Uma fatia *backward* intra-método consiste em todos os vértices que alcançam v por dependência de controle ou de dados. Assim, a partir de v, o *PDG* é percorrido para *trás* pelas arestas de dependência de controle e de dados, recursivamente, obtendo uma cadeia de predecessores, que compõe a fatia *backward* intra-método.

Já no fatiamento *backward* inter-métodos, a estrutura de chamada-retorno de caminhos de execução inter-métodos deve ser levada em consideração. Um algoritmo de fatiamento inter-métodos é desnecessariamente impreciso pelo que é referido na literatura como *calling context problem*. O problema é que quando a travessia de um grafo para o cálculo de uma fatia *backward* "desce" em um procedimento Q que é chamado por um procedimento P, ela vai "subir" para todos os procedimentos

Figura 4.15: Diagrama de Classes do Pacote `ProgramSlicing`.

que chamam Q, não somente P. Isso inclui caminhos de execução inviáveis que entram em Q a partir de P e sai de Q para um procedimento diferente. A travessia de tais caminhos dá origem a fatias imprecisas (Tip, 1995).

Assim, para o cálculo de fatia *backward* intra-método, foi utilizado o algoritmo de Horwitz et al. (1988) de duas etapas, que atravessa um *SDG* aumentado com arestas de sumário (*summary edges*). Dado um vértice como critério s em um método M, a primeira etapa determina todos os vértices que podem alcançar s, sem "descer" em chamadas de procedimento (arestas *parameter-out* não são seguidas). No entanto, esses procedimentos não são ignorados, pois a presença de arestas de sumário de vértices *actual-in* para *actual-out* permite a descoberta de vértices que podem alcançar s somente por meio de chamada de procedimento, somente não descendo no procedimento. A segunda etapa

determina vértices que podem alcançar s de chamadas de procedimentos chamados por P ou de procedimentos chamados por procedimentos que chamam P, "descendo" em todas as chamadas de procedimento anteriormente contornadas. As arestas *call* e *parameter-in* não são seguidas e, assim, a travessia não "sobe" em procedimentos de chamada. As arestas de sumário tornam essas "subidas" desnecessários. O resultado de fatia inter-método consiste no conjunto de vértices identificados nas etapas 1 e 2.

O fatiamento *backward* dinâmico, intra-método ou inter-métodos, usa a ideia de Agrawal e Horgan (1990), que consiste em marcar os vértices de um grafo de dependência que são executados por um conjunto de teste e em calcular a fatia dinâmica usando algoritmos de fatiamento estático, considerando os vértices executados. Então, a partir de um vértice de critério v e da escolha de um teste criado na $SoftVis_{4CA}$, o teste é executado e são utilizados os algoritmos descritos há pouco para calcular fatia *backward* dinâmica intra-método ou inter-métodos, considerando apenas os vértices executados pelo teste escolhido.

4.4.7.2 Fatiamento *Forward*

O fatiamento *forward* é similar ao fatiamento *backward*. O fatiamento *forward* intra-método recebe como entrada um *PDG* e um vértice de critério v. Uma fatia *forward* intra-método consiste em todos os vértices que podem ser alcançados por v por dependência de controle ou de dados. Assim, a partir de v, o *PDG* é percorrido para *frente* pelas arestas de dependência de controle e de dados, recursivamente, obtendo uma cadeia de sucessores, que compõe a fatia *forward* intra-método.

Um algoritmo para fatiamento *forward* inter-métodos pode ser definido em PDG, usando os mesmos conceitos empregados para fatiamento *backward* inter-métodos. Como anteriormente, o elemento-chave é o uso de arestas de sumário devido aos efeitos de chamadas de procedimento (Horwitz et al., 1988). Dado um vértice como critério s em um método M, a primeira etapa segue todas as arestas exceto as arestas *call* e *parameter-in*, não "descendo" para os métodos chamados. A segunda etapa não segue as arestas *parameter-out*, não "subindo" nas chamadas de método.

O fatiamento *forward* dinâmico, intra-método ou inter-métodos, usa, também, a ideia de Agrawal e Horgan (1990) e os algoritmos descritos a pouco para calcular fatia *forward* dinâmica intra-método ou inter-métodos.

4.5 O Modelo de Coordenação

A abordagem visual proposta emprega um modelo de coordenação das seis visões apresentadas. As coordenações são realizadas com base em eventos e propagação, isto é, quando ocorrer um evento em uma visão, a propagação desse evento para as demais visões é realizada. Tal coordenação é baseada na técnica *Brushing-and-Linking* (descrita na Seção 3.3.1), que após a seleção de um elemento em uma visão de origem $v1$ os mesmos (ou relacionados) elementos são simultaneamente destacados em uma outra visão de destino $v2$. Na *Camada de Visualização* (vide Figura 4.1), as coordenações são indicadas pelas setas entre elas (a origem da seta é a visão origem de eventos e o destino da seta é a visão refletida pelo evento na visão origem), permitindo uma exploração visual de diferentes níveis de detalhes.

A implementação das coordenações pertence ao pacote `View.Coordination` (vide Figura 4.16).

Quando ocorre a seleção de um elemento visual em uma visão, o *listener* da visão envia o elemento para a classe `Coordinator`, que obtém os dados representados pelo elemento visual, ou dados relacionados, e envia tais para a classe específica que atualiza a visão de destino. Por exemplo, se uma seleção de elemento deve refletir na *Visão Inter-Instruções*, a classe `AggregateOperations` é chamada pela classe `Coordination` – de maneira similar, as classes `HyperbolicOperations`, `TreemapOperations` e `BytecodeOperations` são chamadas pela classe `Coordinator`.

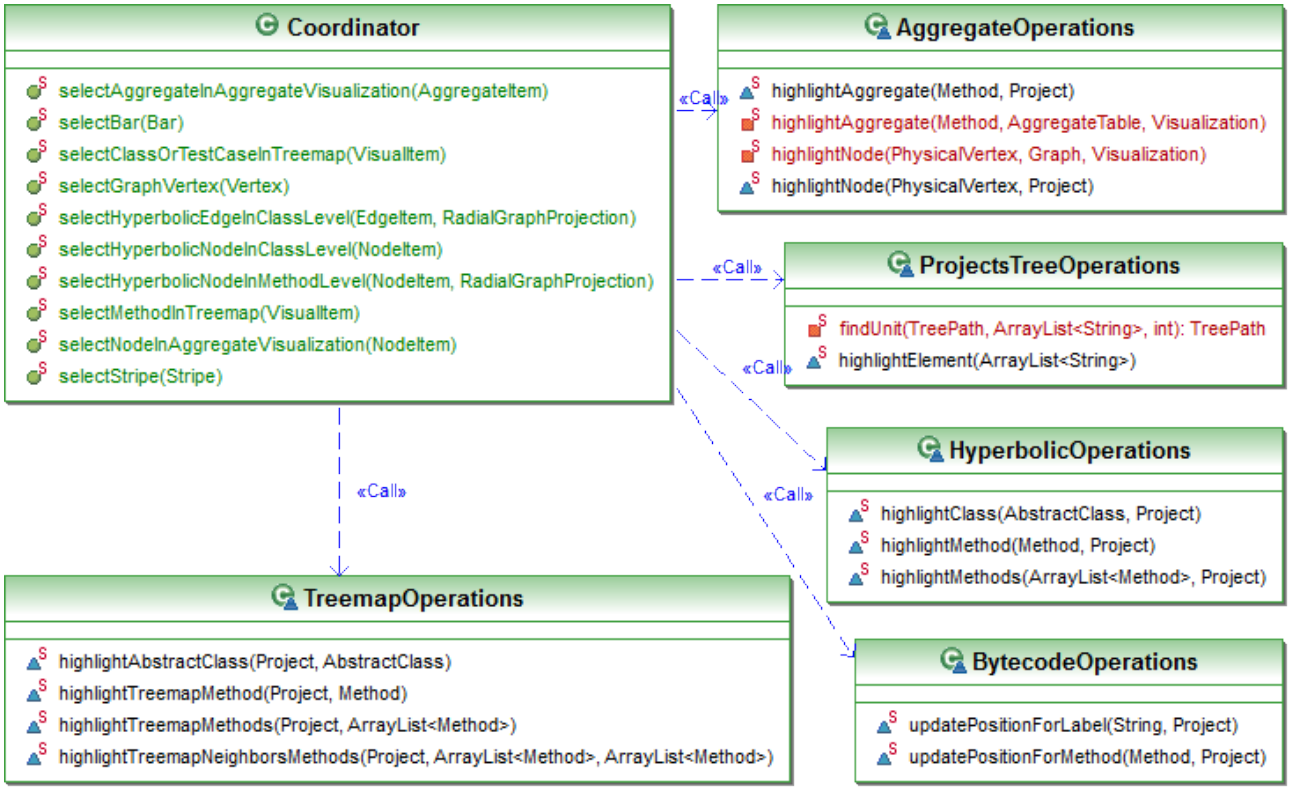


Figura 4.16: Diagrama de Classes do Pacote `View.Coordination`.

As coordenações implementadas, denotadas por $v1 \mapsto \{v2\}$, e as situações em que elas ocorrem, são:

- *Visão Estrutural* $\mapsto$ *Visão Inter-Classes*, *Visão Inter-Métodos*, *Visão de Bytecode*
  - Seleção de classe ou caso de teste: são destacados a classe ou o caso de teste na *Visão Inter-Classes* e os métodos da classe na *Visão Inter-Métodos*;
  - Seleção de método: são destacados o método na *Visão Inter-Métodos*, a classe do método na *Visão Inter-Classes*, e é atualizada a barra de rolagem da *Visão de Bytecode* para o início do código do método;
- *Visão Inter-Métodos* $\mapsto$ *Visão Estrutural*, *Visão Inter-Classes*, *Visão Inter-Instruções*, *Visão de Bytecode*
  - Seleção de método (nó): são destacados o método e os métodos vizinhos do método na *Visão Estrutural*, a classe do método na *Visão Inter-Classes*, o método na *Visão Inter-Instruções*, e é atualizada a barra de rolagem da *Visão de Bytecode* para o início do código do método;

- *Visão Inter-Classes* $\mapsto$ *Visão Estrutural*, *Visão Inter-Métodos*
  - Seleção de classe (nó): são destacados a classe na *Visão Estrutural* e os métodos da classe na *Visão Inter-Métodos*;
  - Seleção de relação entre classes (aresta): são destacados os métodos chamados representados pela aresta na *Visão Inter-Métodos* e na *Visão Estrutural*;
- *Visão Inter-Instruções* $\mapsto$ *Visão Inter-Métodos*, *Visão Estrutural*, *Visão de Bytecode*
  - Seleção de método (região de agregado): é destacado o método na *Visão Inter-Métodos* e na *Visão Estrutural*;
  - Seleção de bloco de instruções bytecode (nó): é atualizada a barra de rolagem da *Visão de Bytecode* para o início do código do bloco de instruções bytecode;
- *Visão de Distribuição de Instruções* $\mapsto$ *Visão de Bytecode*, *Visão Inter-Instruções*, *Visão Inter-Métodos*, *Visão Inter-Classes*
  - Seleção de bloco de instruções bytecode (listra): é atualizada a barra de rolagem da *Visão de Bytecode* para o início do código do bloco de instruções bytecode, e são destacados o bloco de instrução bytecode na *Visão Inter-Instruções* e o método que possui o bloco na *Visão Inter-Métodos*;
  - Seleção de classe (barra): é destacada a classe na *Visão Inter-Classes*.

4.6 Preferências

A ferramenta $SoftVis_{4CA}$ dispõe de opções de preferências para as visualizações, permitindo, assim, que o usuário personalize-as de acordo com suas preferências. As preferências são criadas por classes `PreferencesPanel` que estendem a `JPanel` e criam os componentes necessários para que as preferências possam ser editadas – as preferências da projeção de agregados são implementadas no pacote `View.Aggregate`; as preferências da projeção hiperbólica é implementada no pacote `View.Hyperbolic`; as preferências da projeção treemap é implementada no pacote `View.Treemap`; as preferências da projeção de barras e listras é implementada no pacote `View.BarsAndStripes`; as preferências da escala de cores é implementada no pacote `View.ColorMappingBasedOnCalls`. Por meio da criação de um objeto `JTabbedPane`, as preferências são disponibilizadas em abas – *Aggregate Projection*, *Hyperbolic Projection*, *Treemap Projection*, *Bars and Stripes Projection* e *Color Scale*. Na Figura 4.17 é mostrada uma visão geral das preferências.

As preferências para a *Projeção de Agregados*, são:

- *Node Fill Color*: permite alterar as cores dos diferentes tipos de nós – nós que representam instruções, entradas de métodos, saídas de métodos e parâmetros – bem como as cores de nós quando o *mouse* está sobre e os nós destacados. Além disso, permite a alteração da cor dos nós de instruções executados por um teste e a cor dos nós executados por dois testes ou mais;

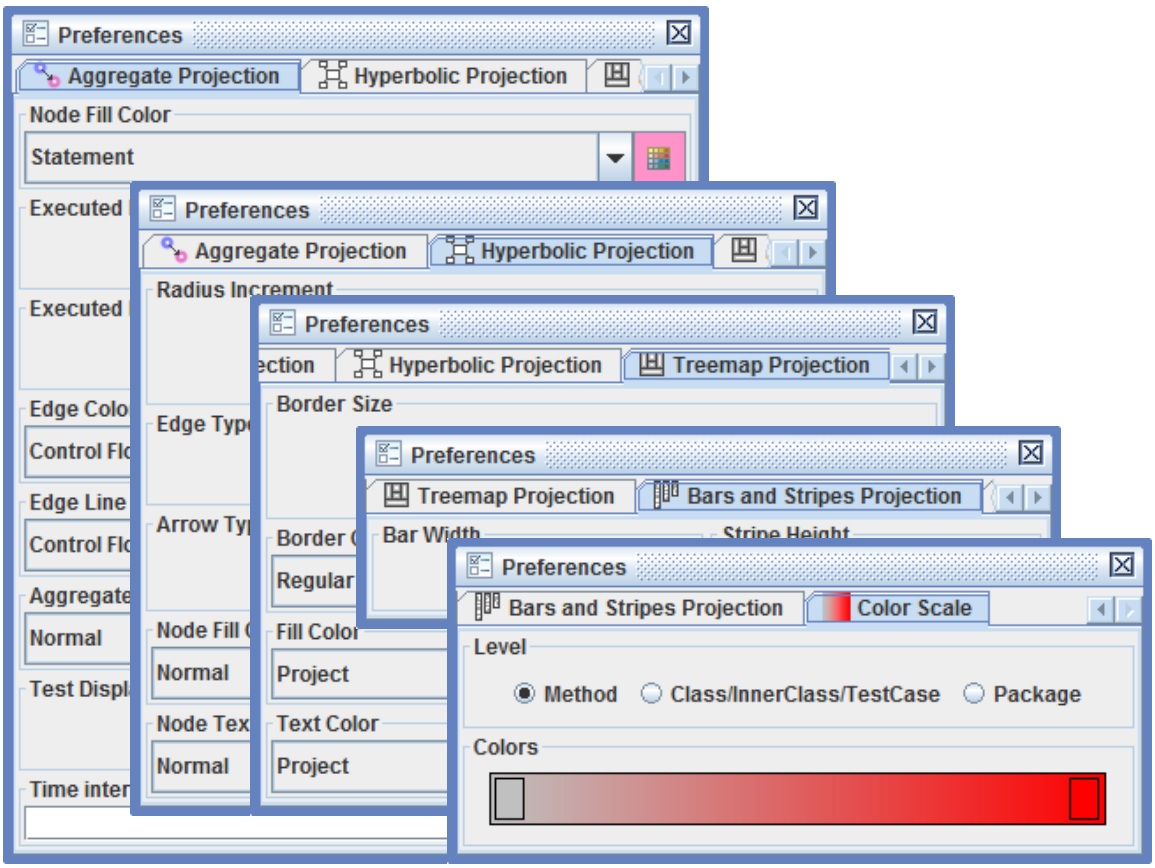


Figura 4.17: Preferências.

- *Executed Node Fill Color*: permite definir se os nós de instruções executadas por um teste serão coloridos pela cor padrão (definida em *Node Fill Color*) ou se serão coloridos pela cor atribuída ao teste que executou tais nós;
- *Executed Node More Than Once Fill Color*: permite definir se os nós de instruções executadas por dois testes ou mais serão coloridos pela cor padrão (definida em *Node Fill Color*) ou se serão coloridos pela cor atribuída ao último teste visualizado que executou tais nós;
- *Edge Color*: permite alterar as cores dos diferentes tipos de arestas – arestas que representam fluxo de controle regular e de exceção, dependência de controle intra-método e inter-método, dependência de dados intra-método e inter-método – bem como a cor quando o *mouse* está sobre a aresta;
- *Edge Line Type*: permite alterar os traços (sólido, pontilhado e tracejado) dos diferentes tipos de arestas – arestas que representam fluxo de controle regular e de exceção, dependência de controle intra-método e inter-método, dependência de dados intra-método e inter-método;
- *Aggregate Stroke Color*: permite alterar a cor das bordas de agregados normal, agregado com o *mouse* sobre e agregado destacado;
- *Test Display Option*: permite definir a visualização de testes, podendo ser normal, em que todos os nós executados por um determinado teste são coloridos, ou podendo ser animada, em que os nós são coloridos de acordo com a ordem de execução;

- *Time interval of executed instruction highlight*: permite definir o tempo utilizado na animação entre a coloração de um nó para o outro.

As preferências para a *Projeção Hiperbólica*, são:

- *Radius Increment*: permite alterar a distância entre os nós;
- *Edge Type*: permite alterar o tipo de traço das arestas, que pode ser traço reto ou curvo;
- *Arrow Type*: permite alterar o tipo da seta das arestas, que pode ser sem seta, seta para frente ou seta para trás;
- *Node Fill Color*: permite alterar as cores dos diferentes tipos de nós – nós normais, em foco e vizinhos do nós em foco – bem como as cores de nós pesquisados e quando o *mouse* está sobre;
- *Node Text Color*: permite alterar as cores dos textos de diferentes tipos de nós – nós normais, em foco e vizinhos do nós em foco – bem como as cores de nós pesquisados e quando o *mouse* está sobre.

As preferências para a *Projeção Treemap*, são:

- *Border Size*: permite alterar o tamanho da borda dos retângulos aninhados;
- *Border Color*: permite alterar a cor da borda de retângulos normais, com o *mouse* sobre, pesquisados ou destacados;
- *Fill Color*: permite alterar as cores dos diferentes tipos de retângulos – retângulos que representam projeto, pacote, classe/classe aninhada, caso de teste e método – bem como as cores de retângulos pesquisados e destacados;
- *Text Color*: permite alterar as cores dos textos de diferentes tipos de retângulos – retângulos que representam projeto, pacote, classe/classe aninhada, caso de teste e método.

As preferências para a *Projeção de Barras e Listras*, são:

- *Bar Width*: permite alterar a largura das barras;
- *Stripe Height*: permite alterar a altura das listras.

As preferências para a *Escala de Cores*, são:

- *Level*: permite alterar o nível da escala de cores, podendo ser em nível de método, de classe/classe aninhada/caso de teste e de pacote;
- *Colors*: permite alterar as cores inicial e final da escala de cores.

4.7 Persistência

O ambiente da ferramenta, isto é, o conjunto de projetos carregados e as preferências modificadas, é mantido em um arquivo XML. Todas as operações que modificam o ambiente são monitoradas e refletidas no arquivo automaticamente. Quando a ferramenta é iniciada, o ambiente é carregado a partir do arquivo XML automaticamente.

A implementação da persistência pertence ao pacote `Persistence` (vide Figura 4.18). A classe `XMLDocument` representa o documento XML. O aspecto `Persistence` monitora o modelo de entidades e as preferências das visões, capturando as modificações a serem persistidas. No caso de uma modificação, o aspecto envia a modificação para a classe responsável por ela. Se a alteração é no modelo de entidades, por exemplo, a classe que o aspecto chama é a `EntitiesOperations`, e então é realizada a gravação da informação no arquivo XML. Quando a ferramenta é iniciada, a classe `EnvironmentLoader` lê todo o arquivo XML e carrega as informações persistidas na ferramenta.

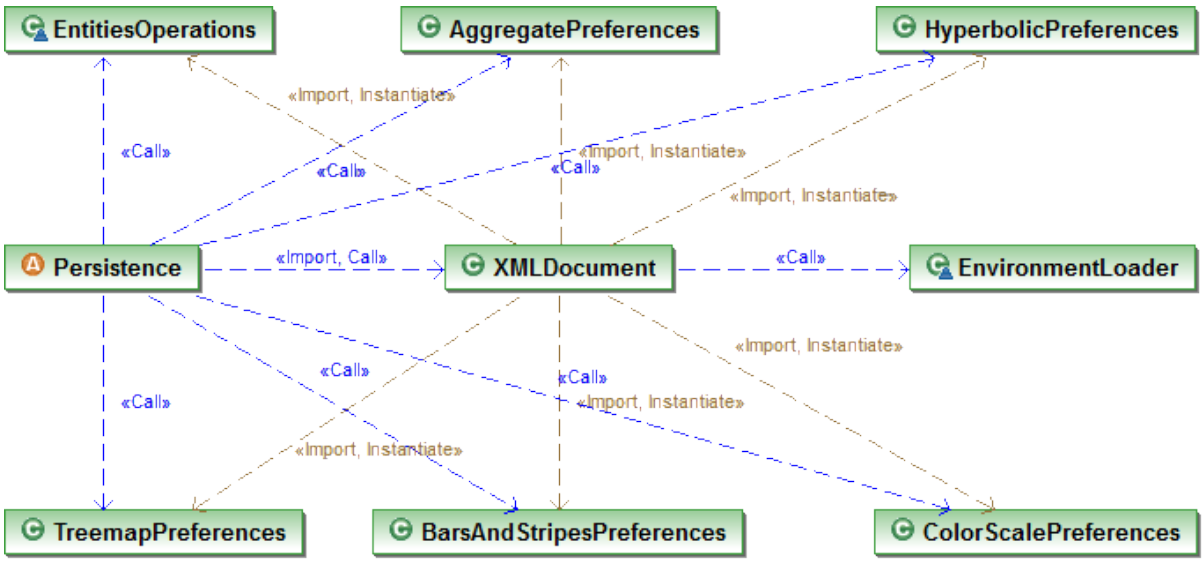


Figura 4.18: Diagrama de Classes do Pacote `Persistence`.

4.8 Considerações Finais

O modelo de coordenação proposto neste trabalho, para a visualização de características de programas desenvolvidos em *Java* juntamente com valores da métrica *fan-in* e com fatias, foi apresentado. Além disso, a arquitetura da ferramenta que implementa o modelo de coordenação proposto, que é dividida em três camadas, foi ilustrada, bem como foi explicada a função de cada camada. Os módulos de implementação de cada camada da arquitetura foram explicados, indicando sua função e as principais classes de sua implementação. As representações visuais obtidas com a ferramenta $SoftVis_{4CA}$ são apresentadas e explicadas no próximo Capítulo.

CAPÍTULO 5

Representações Visuais e Um Estudo Piloto

Os resultados visuais obtidos com a ferramenta $SoftVis_{4CA}$ são as representações visuais. Inicialmente, neste capítulo são apresentadas e explicadas cada representação visual separadamente (Seção 5.1). Uma estratégia para a utilização das representações visuais é definida para auxiliar a identificação de interesses transversais e definição de *pointcuts* (a estratégia é apresentada na Seção 5.2). Um exemplo de uso da estratégia é apresentado (Seção 5.3) e por fim considerações sobre análise dos resultados visuais obtidos são apresentadas (Seção 5.4).

O programa utilizado como exemplo para a apresentação das representações visuais obtidas e para a demonstração da utilização da ferramenta $SoftVis_{4CA}$ chama-se *elevator*, no qual é realizada a simulação de um elevador. O programa é orientado a objetos implementado em *Java*, que possui 12 classes e 100 métodos. O diagrama de classes do programa é ilustrado na Figura 5.1. A implementação do programa está disponível em um repositório[1] para fins de pesquisa em Engenharia de Software (Do et al., 2005).

5.1 Representações Visuais

Nas Figuras 5.2 e 5.3 são mostradas as visões hiperbólicas em nível de classe e de método, respectivamente. Os nós representam unidades (classes ou métodos), e as arestas direcionadas representam as chamadas entre elas. A projeção hiperbólica mostra um nó em foco (escolhido pelo usuário) maior do que os outros, posicionado no centro do espaço de projeção e colorido em roxo. Os nós vizinhos do nó em foco são coloridos em rosa, e os outros nós não são coloridos. Tais visões ajudam a ter evidências de possíveis interesses transversais por meio da visualização e da análise de resultados da métrica *fan-in*, e se o usuário decidir refatorar métodos para aspectos, ajudam a identificar quais unidades são afetados – analisando as chamadas mostradas em nível de método é possível observar o método a ser considerado na criação de *advices* e os que devem ser entrecortados.

Nas Figuras 5.4 e 5.5(a) são mostrados os dois tipos de *Visão Inter-Instruções*: visão do grafo

[1] http://sir.unl.edu/portal/index.php

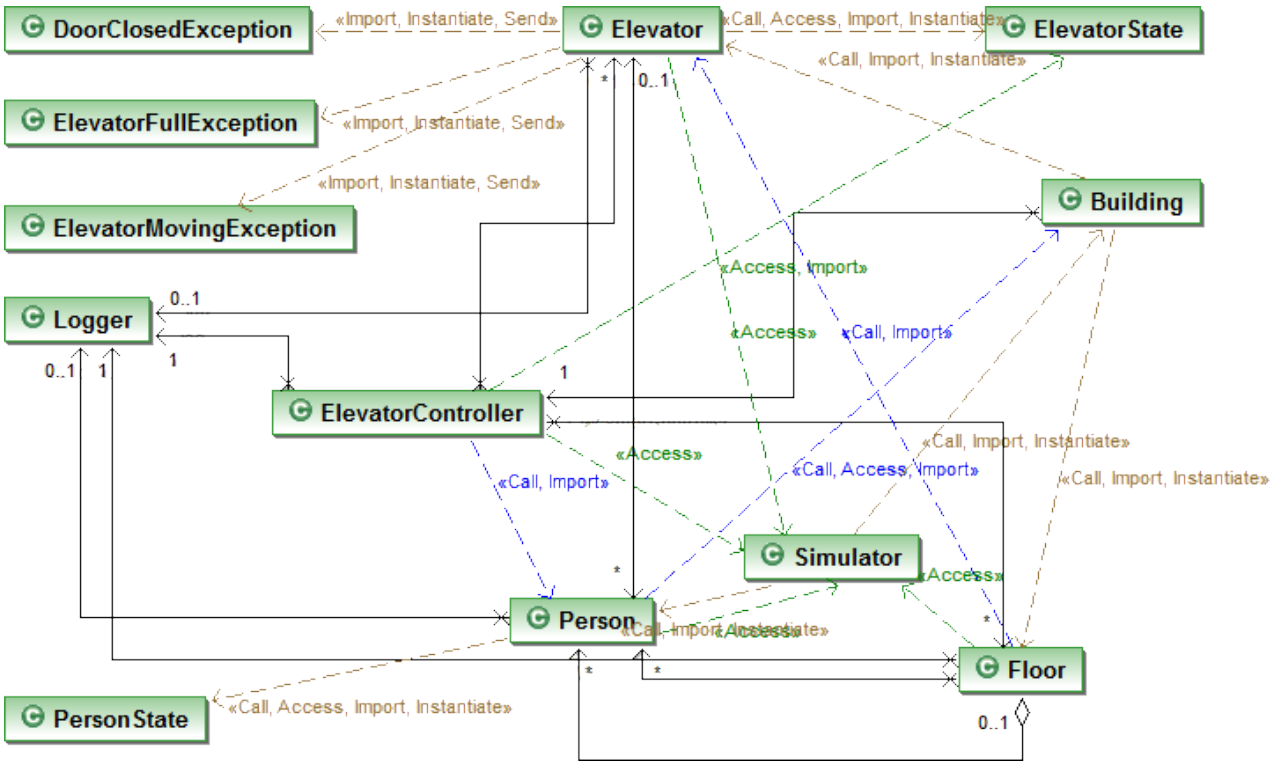


Figura 5.1: Diagrama de Classes do programa *elevator*.

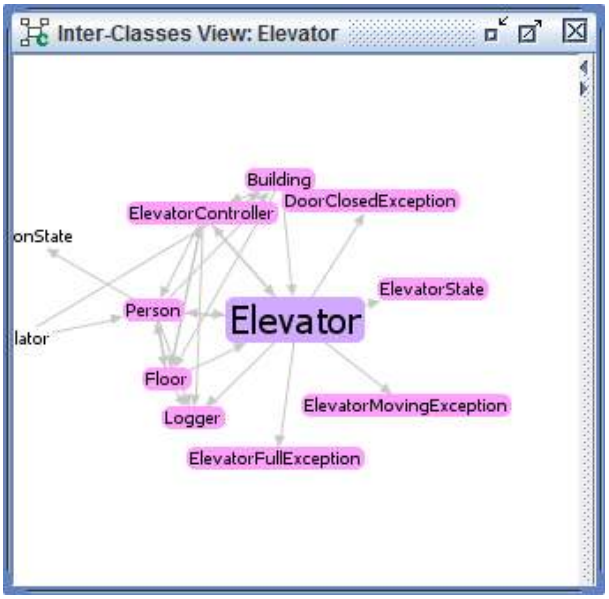


Figura 5.2: *Visão Inter-Classes*.

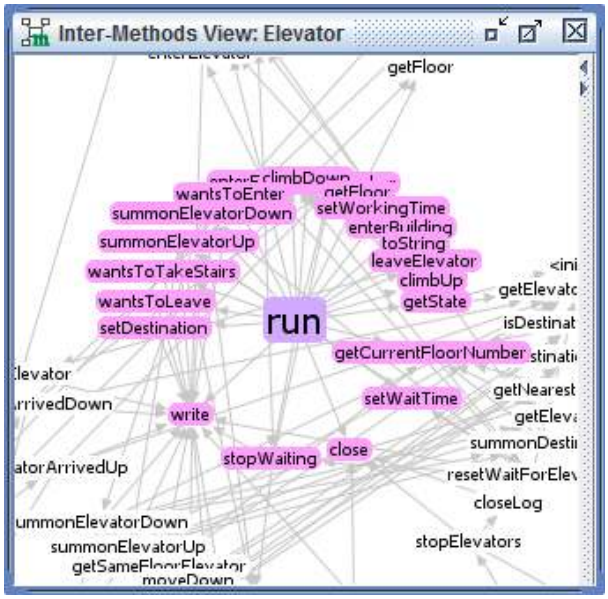


Figura 5.3: *Visão Inter-Métodos*.

de um método (intra-método) e visão de um grafo combinado (inter-método), respectivamente. As instruções de programa são representadas por nós – cada nó é um grupo de instruções bytecode. Além disso, os nós são agregados por métodos, representados por polígonos curvos (nomeado *região de agregado*) coloridos por uma cor específica para cada método (foi utilizada transparência por causa da sobreposição de regiões de agregados). Existem dois tipos de nós: nós físicos e nós virtuais. Os nós físicos são instruções de programa (coloridos em rosa). Os nós virtuais são introduzidos para representar entradas de métodos (coloridos em verde escuro), saídas de métodos (coloridos em verde claro) ou parâmetros (coloridos em preto). As arestas representam fluxo de controle (linhas contínuas

coloridas em preto), exceção (linhas tracejadas coloridas em preto), dependência de controle (linhas contínuas representam intra-método e as linhas tracejadas representam inter-método, ambas coloridas em azul) ou dependência de dados (linhas contínuas representam intra-método e as linhas tracejadas representam inter-método, ambas coloridas em verde). A *Visão Inter-Instruções* permite a exploração visual em nível de instrução.

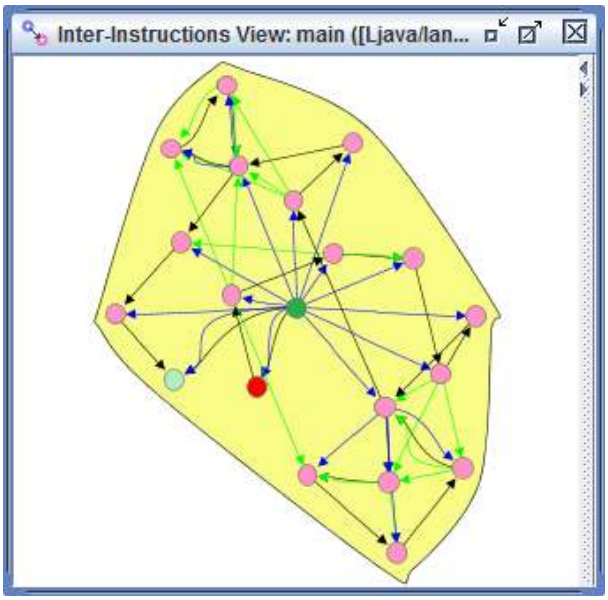
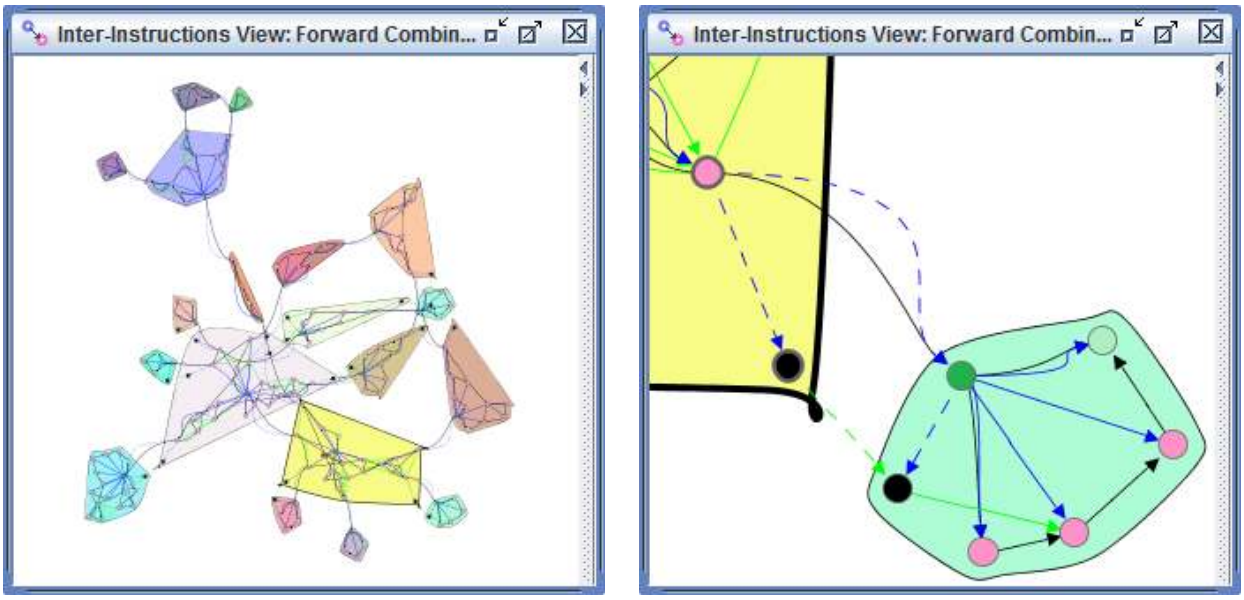


Figura 5.4: *Visão Inter-Instruções* – Intra-método.



(a) Visão Geral. (b) Parte do Grafo com *Zoom*.

Figura 5.5: *Visão Inter-Instruções* – Inter-métodos.

Na Figura 5.6 é mostrada a *Visão Estrutural*. As unidades do programa são representadas por retângulos aninhados. O retângulo externo representa o programa todo (raiz), e os retângulos internos representam pacotes, classes, métodos e casos de testes. Os retângulos são coloridos usando um

gradiente linear simples de cinza (cinza escuro representa o programa todo). Cada retângulo folha representa um método e o seu tamanho é proporcional a quantidade de instruções bytecode.

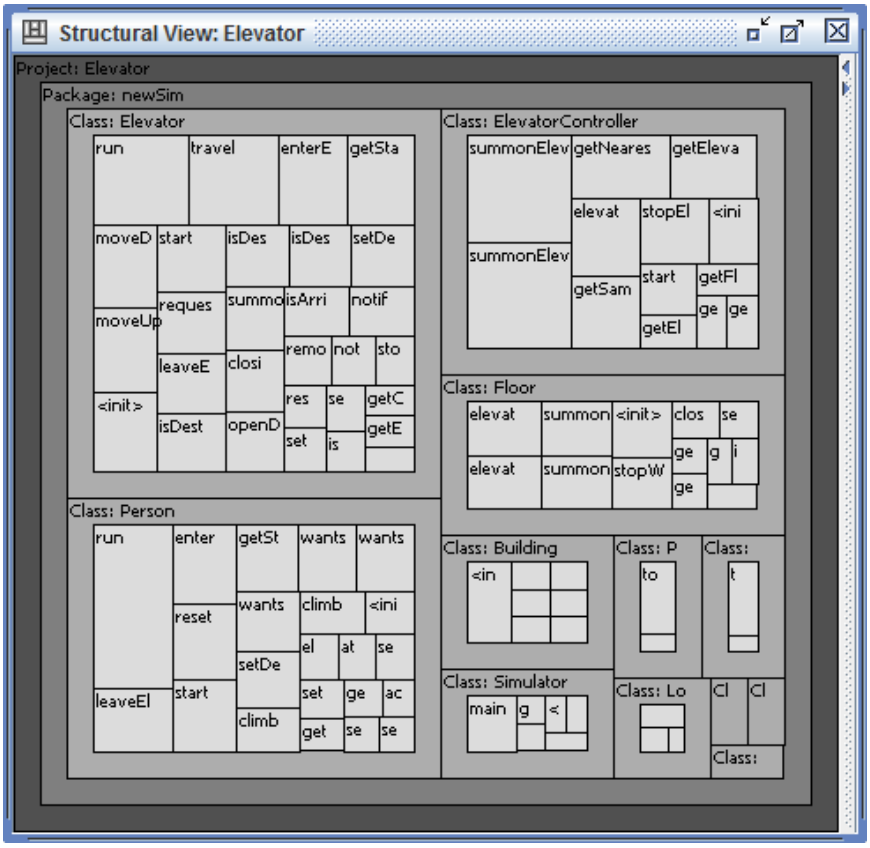


Figura 5.6: *Visão Estrutural*.

Na Figura 5.7 é mostrada a *Visão de Distribuição de Instruções*. As barras representam classes – a altura de cada barra é proporcional a quantidade de instrução bytecode da classe. As listras representam instruções bytecode usando cores diferentes atribuidas para cada conjunto de instruções. Essa visão provê uma visão geral de como um conjunto de instruções é distribuído em múltiplas classes. Como pode ser observado, somente as barras estão visíveis na Figura 5.7. Isso acontece porque nenhum conjunto de instruções está marcado (ou não foi criado) para ser visualizado.

Adicionalmente, a ferramenta $SoftVis_{4CA}$ permite a visualização de instruções bytecode de programa, usando duas rolagens sincronizadas – a rolagem da direita para visão geral e a rolagem da esquerda para detalhes – como mostrado na Figura 5.8. Na rolagem da direita os retângulos representam um conjunto de instruções agrupadas por *labels* de bytecode de programa. Na rolagem da esquerda são mostrados blocos de instruções bytecode de retângulos selecionados na rolagem da direita, agrupadas pelo método que as possuem. Similar à *Visão Inter-Instruções*, cada método é mostrado usando uma cor diferente e o mapeamento de cores é o mesmo da *Visão Inter-Instruções*.

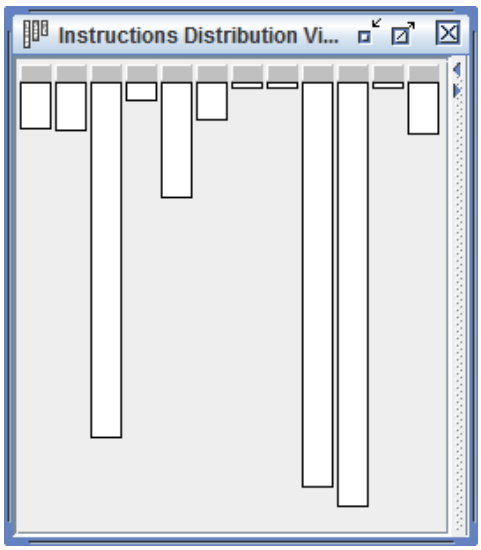


Figura 5.7: *Visão de Distribuição de Instruções*.

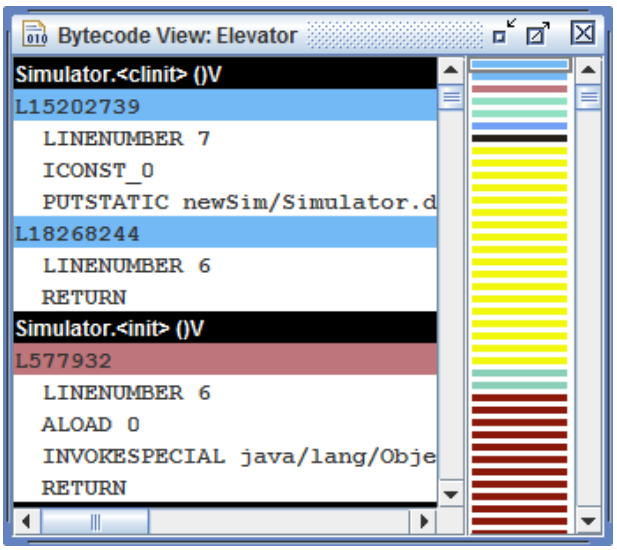


Figura 5.8: *Visão de Bytecode*.

5.2 Abordagem Estratégica *Top-down*

Existe um conjunto significativo de pesquisas sobre compreensão de programa em termos de representações e processos mentais para a criação dessas representações Zhang (2007). Neste trabalho é definida uma estratégia baseada na abordagem *top-down*. O processo de compreensão *top-down* inicia com uma hipótese geral sobre o propósito do programa enquanto detalhes são negligenciados. A hipótese geral é refinada considerando *beacons* (isto é, partes do código fonte que indicam ocorrências de certas estruturas ou operações Brooks (1978)), que determinam o propósito do programa. Um vez que uma hipótese é formulada, o desenvolvedor a avalia olhando para detalhes e refinando sua hipótese gradualmente em uma hierarquia, formulando hipóteses subsidiárias (Feigenspan, 2011, Zhang, 2007).

A estratégia para a utilização das representações visuais da $SoftVis_{4CA}$ é mostrada na Figura 5.9, em que as representações visuais são divididas em três níveis de detalhe – alto, intermediário e baixo – e o fluxo de setas indica a abordagem *top-down*. Primeiro, o usuário pode observar nas representações visuais em alto nível valores da métrica *fan-in* (pacotes e classes), formulando hipóteses iniciais. Pelo modelo de coordenação, o usuário pode refinar a hipótese inicial em um nível intermediário (métodos), para observar e analisar as relações de chamadas de métodos e seus valores *fan-in*. Finalmente, detalhes podem ser observados no nível baixo, como instruções e suas dependências, para ajudar a descrever *pointcuts*.

5.3 Utilizando as Representações Visuais por meio da Estratégia

A frequência de chamadas de métodos (com base na métrica *fan-in*) é calculada em nível de pacote, classe/caso de teste e método, a fim de criar o gradiente de cores mencionados no Capítulo anterior (Seção 4.4.6). Nas Figuras 5.10(a) e 5.10(b) são ilustradas as escalas de cores em nível de classe e de método, respectivamente. Inicialmente, o ponteiro da escala de cores marca o valor

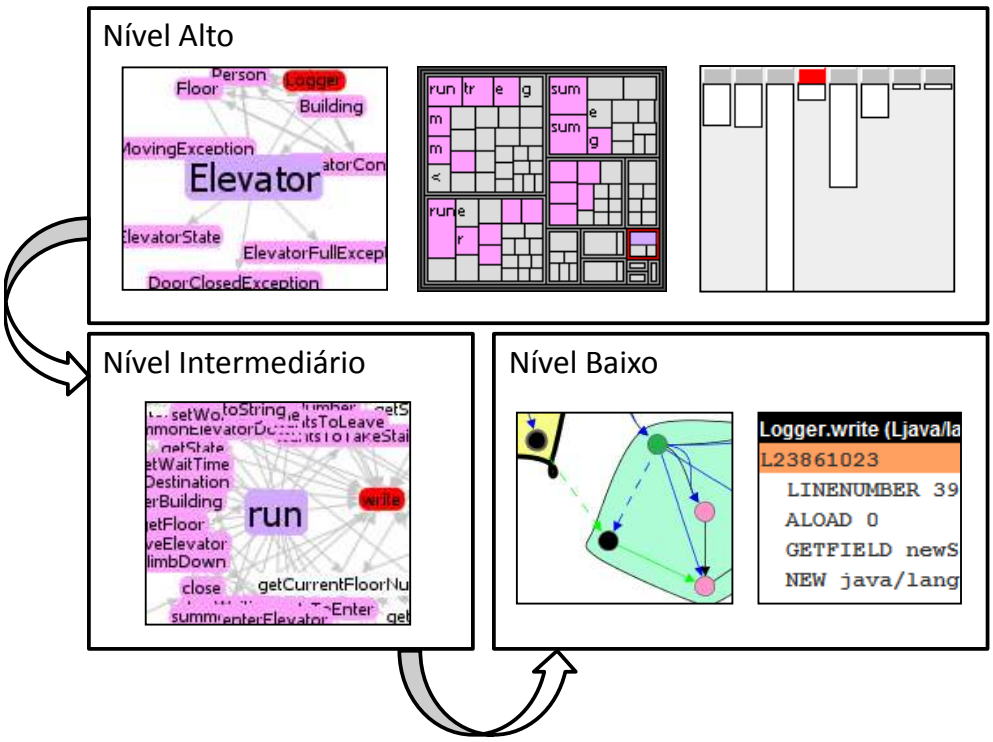


Figura 5.9: Estratégica *top-down* para a utilização das representações visuais.

máximo existente na escala de cores, como na Figura 5.10(a). Se o usuário mudar o ponteiro de lugar, a escala de cores move o ponteiro, automaticamente, para o próximo valor existente na escala de cores – na Figura 5.10(b) o segundo maior valor existente é o 17. Assim, a escala de cores pode ser utilizada para estabelecer um valor limiar (*threshold*) para frequência de chamada a fim de descartar unidades não relevantes.

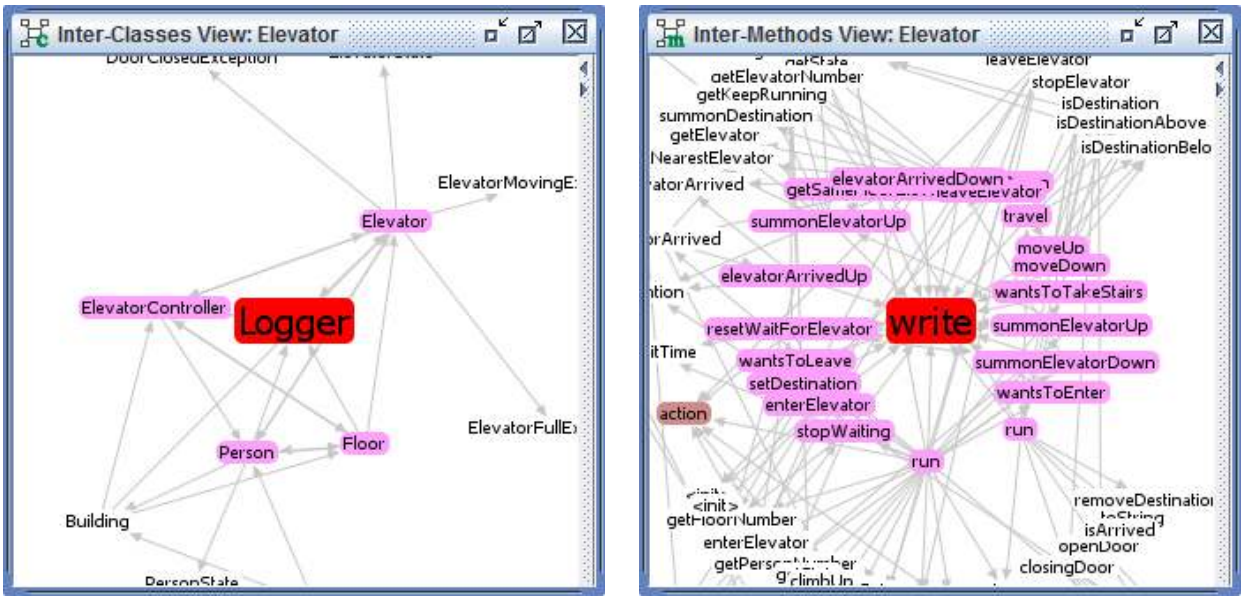
(a) Em nível de classe. (b) Em nível de método.

Figura 5.10: Escalas de Cores.

Na *Visão Inter-Classes* (Figura 5.11(a)) é mostrada a classe mais chamada (`Logger`) destacada em vermelho, isto é, a visão está sendo refletida pela escala de cores da Figura 5.10(a). O usuário pode refinar os resultados da *Visão Inter-Classes* e métodos são destacados na *Visão Inter-Métodos*. Em nível de método, também é possível usar uma outra escala de cores para destacar métodos com alta frequência de chamada. Na Figura 5.11(b) são coloridos os métodos `write` e `action` (foi considerado o valor 17 como frequência mínima[2], como mostrado na Figura 5.10(b)). Baseado nisso, o usuário pode observar explicitamente a frequência de chamadas para cada método filtrado e decidir se ele pode ser considerado um candidato a interesse transversal.

A *Visão Inter-Métodos* tem um papel fundamental no modelo de coordenação. Pela seleção de um método de interesse, é possível ver a dependência de sua classe na *Visão Inter-Classes*, e também é possível observar o método selecionado e os relacionados (chamados e chamadores) na estrutura

[2]A frequência mínima é definida empiricamente, considerando o código fonte sob análise.



(a) *Visão Inter-Classes*. (b) *Visão Inter-Métodos*.

Figura 5.11: Visões refletidas pelas escalas de cores.

hierárquica do programa mostrado na *Visão Estrutural*. A *Visão Estrutural* não é decisiva para identificação de candidato a interesse transversal, mas é interessante ter uma visão geral da estrutura do programa e observar as unidades de código (classes e métodos) que podem ser afetados por refatoração. Na Figura 5.12 é mostrada a estrutura hierárquica destacando o método selecionado na Figura 5.11(b) (sua classe em vermelho) e os métodos relacionados em rosa. Em adicional, detalhes de dependência de dados e de controle de instruções de método selecionado podem ser analisados usando a *Visão Inter-Instruções*, bem como as instruções bytecode na *Visão de Bytecode*.

Se um método é considerado candidato a ser reimplementado em aspecto, é necessário a análise em nível de instruções para a captura de contextos de *join points* que devem ser entrecortados por aspectos. A *Visão Inter-Instruções* mostra grafos em nível de instrução. Cada método tem seu próprio grafo, e grafos podem ser combinados por instruções de chamadas de método de ou para um método selecionado (*forward* ou *backward*). Os grafos de métodos são utilizados para criar fatias intra-método, e grafos combinados são utilizados para criar fatias inter-métodos. As fatias podem ser estáticas ou dinâmicas, de ou para um nó selecionado (fatia *forward* ou *backward*).

Nas Figuras 5.14 e 5.15 é mostrado uma fatia estática *backward* gerada a partir do grafo combinado *backward* baseado no método `write` (o grafo intra-método do método `write` é mostrado na Figura 5.13(a) e o grafo combinado a partir dele é mostrado na Figura 5.13(b)). O método `write` é representado pela *região de agregado* (1). O critério de fatia selecionado é a instrução (2), o nó de entrada para o método `write` é o nó (3) e o parâmetro (4) é um *formal-in* que o método `write` recebeu e é utilizado pelo critério de fatia.

Nas Figuras 5.14 e 5.15 foi utilizado o filtro de distância igual a dois – na Figura 5.14 foi realizada a filtragem a partir do nó de entrada (3) e na Figura 5.15 a partir do nó de parâmetro (4). O filtro de distância foi utilizado por causa da grande quantidade de nós e arestas que formam a fatia. Como na Figura 5.13(b), a fatia criada é difícil de entender, e tal problema de escalabilidade pode ser contor-

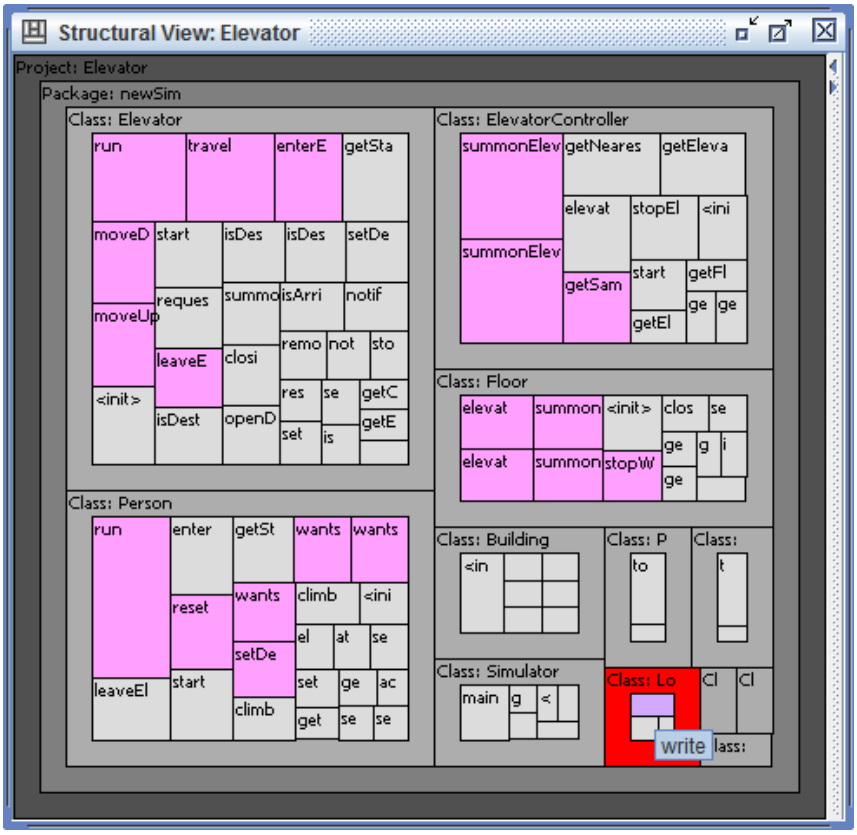
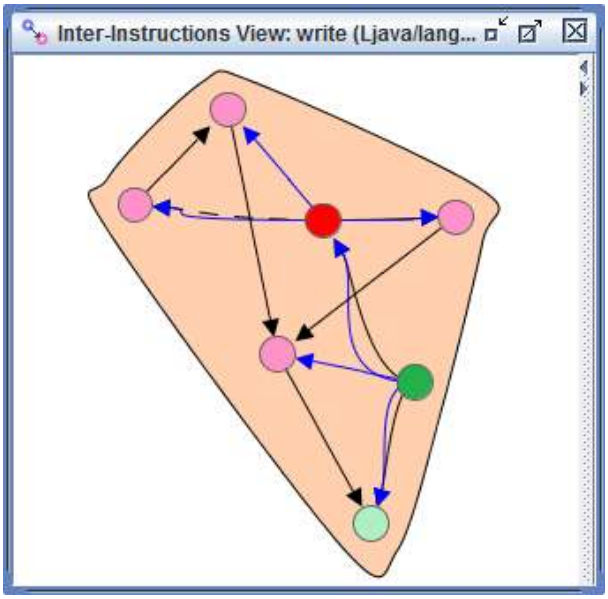
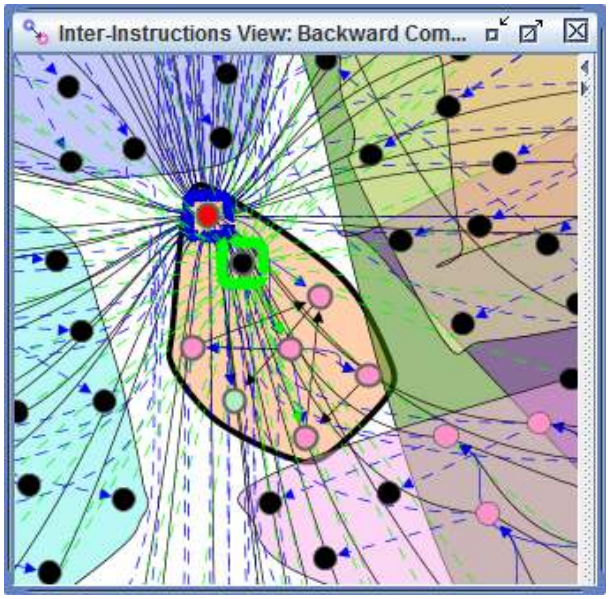


Figura 5.12: *Visão Estrutural* coordenada com a *Visão Inter-Métodos* da Figura 5.11(b).



(a) Grafo do método `write`.



(b) Grafo combinado *backward* baseado no método `write`.

Figura 5.13: *Visões Inter-Instruções*.

nado pela utilização do filtro de distância. Então, pode ser observado: 1) as instruções que chamam diretamente o método dono do critério de fatia analisando as arestas de dependência de controle (em azul) na Figura 5.14; 2) a passagem de parâmetros que são utilizados pelo critério de fatia pela análise das arestas de dependência de dados (em verde) na Figura 5.15. Pela navegação desses nós, o usuário

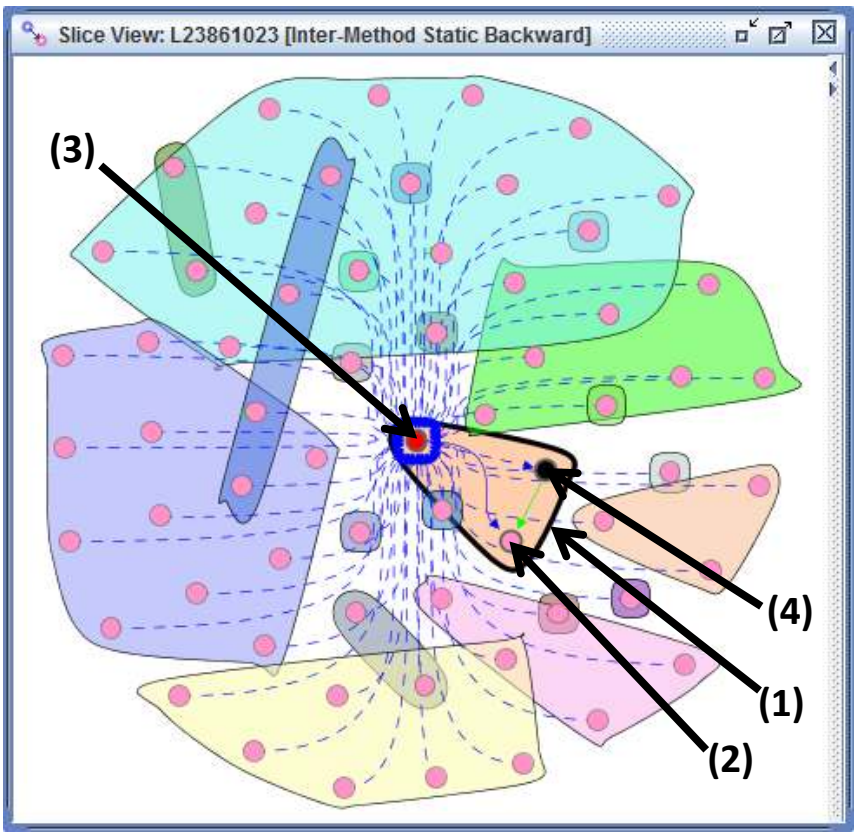


Figura 5.14: Fatia criada, apresentada usando filtro de distância igual a dois com base no nó (3).

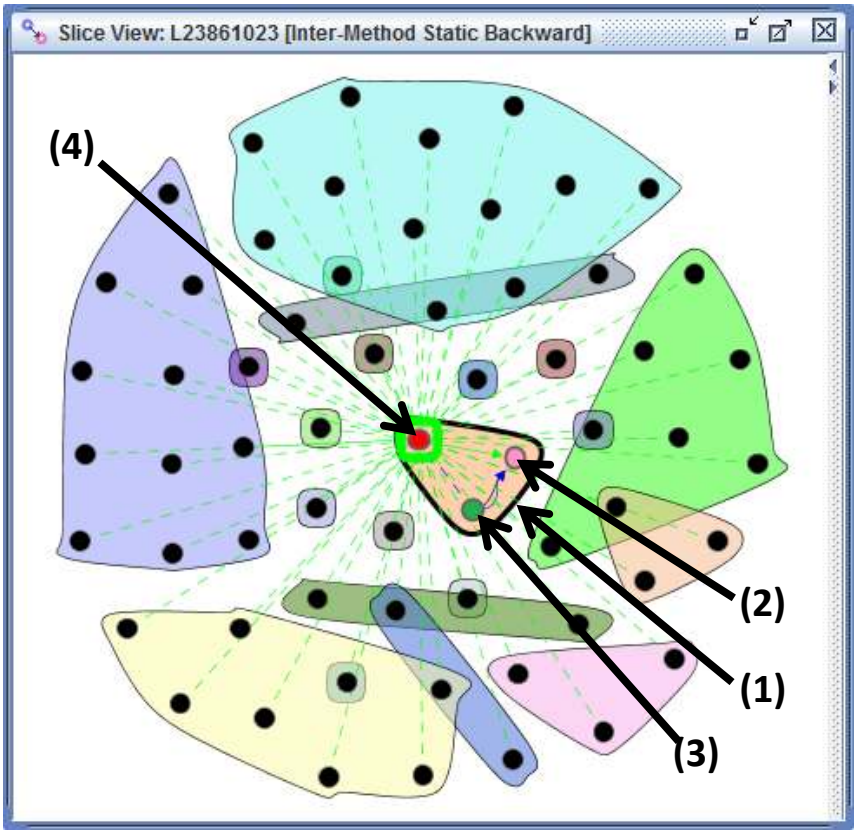


Figura 5.15: Fatia criada, apresentada usando filtro de distância igual a dois com base no nó (4).

pode definir *pointcuts* para capturar *join points* e os dados necessários de seus contextos. Em adicional, uma visão geral de como uma fatia é distribuída em classes é provida pela *Visão de Distribuição de Instruções* na Figura 5.16 (instruções da fatia das Figuras 5.14 e 5.15 são coloridos em vermelho, e a classe `Logger` destacada de acordo com a escala de cores em nível de classe), e na Figura 5.17 é apresentado o código do critério de fatia.

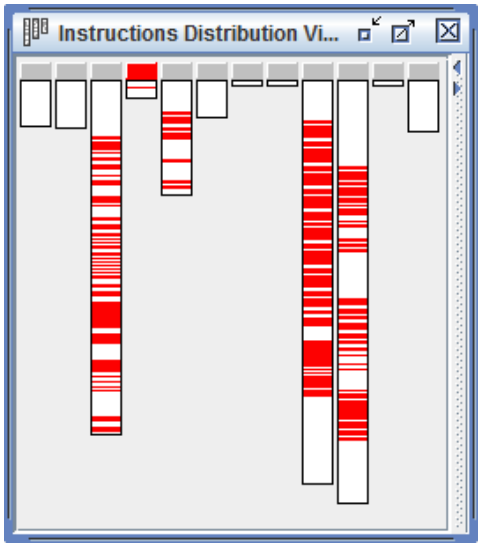


Figura 5.16: *Visão de Barras e Listras* mostrando as instruções que compõem a fatia das Figuras 5.14 e 5.15.

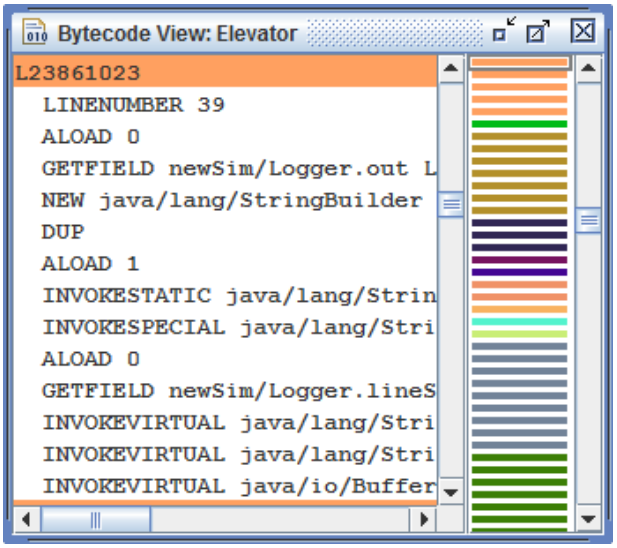


Figura 5.17: *Visão de Bytecode* coordenada com a 5.14, quando o nó que representa o critério de fatia (2) é selecionado.

5.4 Observações e Lições Aprendidas

As *Visões Inter-Classes* e *Inter-Métodos* mostraram ser úteis quanto à visualização de resultados da métrica *fan-in* juntamente com as relações de classes e métodos baseados em chamadas, e a *Visão Inter-Instruções* mostrou ser útil para a análise de pontos que devem ser entrecortados por *pointcuts* e de dados que devem ser capturados. Algumas visões não são decisivas para a identificação de candidato a interesse transversal, como a *Visão Estrutural* e a *Visão de Distribuição de Instruções*, mas é interessante ter uma visão geral da estrutura do programa e observar as unidades de código (classes e métodos) que podem ser afetados por refatoração.

Pela interação com as visões, o usuário pode obter informações sobre métodos e decidir sobre refatorar para aspecto ou não. Métodos considerados candidatos a interesses transversais, suas classes e métodos chamadores, e como eles estão relacionados devem ser analisados em alto e baixo nível. Em baixo nível, o usuário pode observar a dependência de controle e de dados, o que é importante para avaliar como um interesses transversal pode ser implementado em um aspecto (por exemplo, o aspecto deve ter acesso aos dados do método?).

Foi observado que a ferramenta ajuda a lidar com limitações de algumas projeções (escalabilidade de técnicas de visualização). As visões que utilizam projeção hiperbólica podem apresentar uma grande quantidade de dados (classes e métodos), resultando em nós sobrepostos. Isso pode ser contornado modificando a preferência de incremento de raio e utilizando filtro de distância para ocultar os nós que não são de interesse para a exploração visual. De maneira similar, pode ser difícil com-

preender uma *Visão Inter-Instruções* quando o grafo de instruções possui muitos vértices e, para lidar com isso, filtros de distância também podem ser utilizados, bem como a ocultação de tipos de nós e de tipos de arestas, permitindo que o usuário foque na análise de seu interesse.

Adicionalmente, foram observadas algumas considerações sobre fatiamento de programa. Um algoritmo de fatiamento de programa constrói, a partir de instruções de um programa, alguma representação alternativa para calcular fatias. Desde que uma fatia consiste em todas as instruções que podem afetar ou podem ser afetadas pelo critério de fatia, independentemente da representação intermediária gerada, as dependências de controle e de dados devem ser calculadas. Assim, fatiamento de programa pode ajudar na definição de *pointcuts*, que é uma etapa do processo de refatoração para aspectos.

5.5 Considerações Finais

As representações visuais mostradas foram obtidas por meio da implementação da ferramenta $SoftVis_{4CA}$, que implementa a abordagem visual apresentada no Capítulo 4. Para auxiliar a identificação de interesses transversais e a definição de *pointcuts* (no caso da reimplementação de um interesse transversal identificado em aspecto), uma estratégia para analisar as representações visuais foi definida. Foi apresentado um estudo de caso, mostrando um uso das representações visuais por meio da estratégia.

Apesar da estratégia definida para a utilização e a análise das visões ser *top-down*, o usuário pode definir a sua própria estratégia, como por exemplo, a estratégia *bottom-up*, analisando as instruções de um programa e agrupando-as em partes semânticas. No entanto, a abordagem *bottom-up* pode não ser eficaz para encontrar interesses transversais reais. *Logger* é dito ser um interesse transversal bem conhecido, mas em um sistema específico, por exemplo, pode ser que poucos métodos usem métodos que implementa o *logger*. Usando uma estratégia *bottom-up*, o usuário pode identificar o interesse *logger*, mas o efeito da implementação do *logger* pode não ser transversal, e se o *logger* for refatorado para aspecto, o impacto dessa atividade pode não ser significativa.

CAPÍTULO 6

Conclusões

A modularização de interesses transversais ainda é um desafio, pois a implementação de tais interesses tende a ser espalhada por diversos módulos do sistema e emaranhada com outros interesses, o que é um problema de compreensibilidade de sistema de software, e como resultado torna difícil a manutenção. Uma solução é a utilização de mecanismos e abstrações providos pela Programação Orientada a Aspectos, encapsulando a implementação de interesses transversais em uma unidade de código separada (aspecto).

Porém, a evolução de sistemas de software existentes para a tecnologia orientada a aspectos é também um desafio. Duas atividades são envolvidas: a mineração de aspectos, que identifica trechos de códigos em sistemas que potencialmente podem ser reformulados em aspectos, e a refatoração para aspectos, que reformula os trechos de código em aspectos. Diversas técnicas têm sido propostas para a mineração de aspectos, mas ainda com limitações. Segundo alguns autores, uma das causas dessas limitações é a apresentação inadequada dos resultados.

Neste trabalho foi apresentada uma abordagem visual utilizando técnicas de visualização e de interação para a apresentação de resultados de técnicas de mineração de aspectos, em um ambiente para exploração visual de características de programas.

6.1 Contribuições

A abordagem visual é a maior contribuição deste trabalho, que inclui seis visões para apresentar diferentes níveis de detalhes de características de programas juntamente com resultados obtidos por *fatiamento de programa* e métrica *fan-in*. As múltiplas visões são coordenadas, permitindo que o usuário relacione dados selecionados em uma visão com outra, tornando a exploração visual de programa mais eficaz.

A ferramenta $SoftVis_{4CA}$ também é uma contribuição, na qual foi implementado o modelo de coordenação da abordagem visual proposta. Apesar de somente duas técnicas de mineração de aspectos terem sido implementadas, o ambiente provê representações de programas (estruturas de dados) e visualizações em diferentes níveis de granularidade, que podem ser utilizadas para a implementação

de outras técnicas de mineração de aspectos. Tendo um ambiente único para a apresentação de resultados, a questão de difícil comparabilidade dos resultados é minimizada. Mesmo que os resultados sejam em diferentes níveis de granularidade, pela coordenação, é permitido que o usuário relacione ou use em conjunto resultados gerados por diferentes técnicas.

A abordagem *top-down* foi proposta para a utilização de fatiamento de programa juntamente com a métrica *fan-in*. A utilização de fatiamento de programa em mineração de aspectos e refatoração para aspectos foi proposta na literatura recentemente, e o seu uso para mineração de aspectos era inadequado. O programa inteiro deveria ser fatiado e, então, o usuário deveria analisar as fatias para identificar fatias que implementam interesses transversais. Uma contribuição para a utilização de fatiamento de programa, foi definir a abordagem *top-down*, que a partir de resultados gerados por uma técnica de mineração de aspectos em um nível mais alto, no caso a análise de *fan-in*, o usuário pode focar em métodos específicos e visualizar fatias baseados nele. Assim, detalhes são visualizados apenas para a confirmação de interesses transversais e para refatoração para aspectos.

O estudo piloto mostrou que o modelo de coordenação proposto apoia a análise pela exploração de diferentes níveis de detalhe. Além disso, foi concluído que a ferramenta ajuda a lidar com limitações de algumas projeções (escalabilidade de técnicas de visualização), por meio da aplicação de filtros ou modificação de preferências.

6.2 Limitações

Os resultados visuais e as observações (lições aprendidas) apresentadas foram obtidos a partir do estudo piloto. No entanto, não é possível obter evidências contundentes por meio de um estudo piloto. Para avaliar a eficácia e a eficiência da abordagem, um experimento controlado está sendo planejado, em que foram definidos os objetivos da avaliação – avaliar o uso da ferramenta $SoftVis_{4CA}$ como um auxílio para a identificação de interesses transversais e para a definição de *pointcuts* para a captura de *join points* e de seus contextos em programas orientados a objetos.

6.3 Trabalhos Futuros

O principal trabalho futuro é a execução do experimento controlado planejado. Os trabalhos futuros adicionais podem ser divididos em três categorias principais: questões de análise das visões, modelo de coordenação da abordagem visual e ferramenta. Em termos da estratégia para análise e utilização das visões providas pela ferramenta, uma lista de diretrizes para uma análise específica dos nós e arestas de grafos apresentados em *Visões Inter-Instruções*, para a definição de *pointcuts*, pode ser útil.

Em relação ao modelo de coordenação e representações visuais, outras técnicas de visualização podem ser adicionadas, tanto para apresentar de uma maneira diferente os mesmos conjuntos de dados obtidos a partir de programas quanto para apresentar outros conjuntos de dados. A adição de outras técnicas de visualização podem ser facilmente acomodadas na ferramenta, já que a sua implementação em camadas separa dados de visualização.

Em relação à ferramenta, diversas melhorias podem ser realizadas. A engenharia reversa de bytecode para código fonte *Java* pode melhorar a análise de programa, uma vez que o usuário é fa-

miliarizado com o código fonte. Além disso, outras técnicas de mineração de aspectos podem ser implementadas, sendo seus resultados mapeados em itens visuais das visões já existentes. Por exemplo, a *análise de link* (Huang et al., 2010) identifica interesses transversais em nível da classe, e os resultados obtidos com essa técnica podem ser mapeados na *Visão Inter-Classes*. Um tipo de técnica de detecção de clones utiliza grafos de dependência em nível de instrução como representação intermediária para obter resultados. A ferramenta constrói grafos de dependência e utiliza a *Visão Inter-Instruções* para a apresentação, tendo suporte, então, para a implementação de detecção de clones e exibição de seus resultados.

6.4 Produção Bibliográfica

Adicionalmente, dois artigos relacionados à abordagem visual foram publicados, ambos em 2013, como segue:

- DELFIM, F. M.; GARCIA, R. E. Multiple Coordinated Views to Support Aspect Mining Using Program Slicing. In: *25th International Conference on Software Engineering and Knowledge Engineering (SEKE'13)*, Boston, USA, June 2013.
- DELFIM, F. M.; GARCIA, R. E. Uma Proposta de Múltiplas Visões Coordenadas para Apoiar Análise de Impacto de Mudança. In: *X Workshop de Manutenção de Software Moderna (WMSWM'13)*, Salvador, Bahia, Julho 2013.

Referências Bibliográficas

ABAIT, E. S.; VIDAL, S. A.; MARCOS, C. A. Dynamic Analysis and Association Rules for Aspects Identification. *Proceedings of the II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP '08)*, p. 31–39, 2008.

AGRAWAL, H.; HORGAN, J. R. Dynamic Program Slicing. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*, New York, NY, USA: ACM, 1990, p. 246–256.

AHLBERG, C.; WISTRAND, E. Ivee: An information visualization and exploration environment. In: *Proceedings of the 1995 IEEE Symposium on Information Visualization (INFOVIS '95)*, Washington, DC, USA: IEEE Computer Society, 1995, p. 66–73.

ANKERST, M. *Visual Data Mining*. Tese de Doutoramento, Universidade de Munique, 2000.

ARCOVERDE, R.; GARCIA, A.; FIGUEIREDO, E. Understanding the longevity of code smells: preliminary results of an explanatory survey. In: *Proceedings of the 4th Workshop on Refactoring Tools (WRT '11)*, New York, NY, USA: ACM, 2011, p. 33–36.

ATKINSON, D. C.; KING, T. Lightweight Detection of Program Refactorings. In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC '05)*, Washington, DC, USA: IEEE Computer Society, 2005, p. 663–670.

BALL, T.; EICK, S. G. Software Visualization in the Large. *IEEE Computer*, v. 29, n. 4, p. 33–43, 1996.

BREU, S.; KRINKE, J. Aspect Mining Using Event Traces. In: *19th IEEE International Conference on Automated Software Engineering (ASE '04)*, Washington, DC, USA: IEEE Computer Society, 2004, p. 310–315.

BROOKS, R. Using a Behavioral Theory of Program Comprehension in Software Engineering. In: *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*, Piscataway, NJ, USA: IEEE Press, 1978, p. 196–201.

BRUNTINK, M.; VAN DEURSEN, A.; D'HONDT, M.; TOURWÉ, T. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In: *Proceedings of the 6th International Conference on Aspect-oriented Software Development (AOSD '07)*, New York, NY, USA: ACM, 2007, p. 199–211.

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B., eds. *Readings in information visualization: Using vision to think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

CARNEIRO, G. F. *Usando Medição de Código Fonte para Refactoring*. Dissertação de mestrado, Universidade Salvador, 2003.

CECCATO, M.; MARIN, M.; MENS, K.; MOONEN, L.; TONELLA, P.; TOURWÉ, T. Applying and combining three different aspect Mining Techniques. *Software Quality Control*, v. 14, n. 3, p. 209–231, 2006.

CLIFTON, C.; LEAVENS, G. T.; NOBLE, J. MAO: Ownership and Effects for More Effective Reasoning About Aspects. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '07)*, Springer-Verlag, 2007, p. 451–475.

D'ARCE, A. F.; GARCIA, R. E.; CORREIA, R. C. M.; ELER, D. M. Coordination Model to Support Visualization of Aspect-Oriented Programs. In: *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE '12)*, 2012, p. 168–173.

DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

DO, H.; ELBAUM, S. G.; ROTHERMEL, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, v. 10, n. 4, p. 405–435, 2005.

DROZDZ, M.; KOURIE, D. G.; WATSON, B. W.; BOAKE, A. Refactoring Tools and Complementary Techniques. In: *Proceedings of the IEEE International Conference on Computer Systems and Applications (AICCSA '06)*, Washington, DC, USA: IEEE Computer Society, 2006, p. 685–688.

EICK, S. G.; STEFFEN, J. L.; ERIC E. SUMNER, J. Seesoft – A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, v. 18, n. 11, p. 957–968, 1992.

ELER, D. M. *Múltiplas visões coordenadas para exploração de mapas de similaridade*. Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação (ICMC-USP), 2011.

FEIGENSPAN, J. Program Comprehension of Feature-Oriented Software Development. In: *Proceedings of the International Doctoral Symposium on Empirical Software Engineering (IDoESE)*, 2011.

FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions Programming Languages and Systems*, v. 9, n. 3, p. 319–349, 1987.

FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M., eds. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

FONTANA, F. A.; SPINELLI, S. Impact of refactoring on quality code evaluation. In: *Proceedings of the 4th Workshop on Refactoring Tools (WRT '11)*, New York, NY, USA: ACM, 2011, p. 37–40.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 464 p., 1999.

FRAWLEY, W. J.; PIATETSKY-SHAPIRO, G.; MATHEUS, C. J. Knowledge discovery in databases: An overview. In: *Knowledge Discovery in Databases*, AAAI/MIT Press, p. 1–30, 1991.

FREDRIKSON, A.; NORTH, C.; PLAISANT, C.; SHNEIDERMAN, B. Temporal, Geographical and Categorical Aggregations Viewed through Coordinated Displays: A Case Study with Highway Incident Data. In: *Proceedings of the 1999 Workshop on New Paradigms in Information Visualization and Manipulation in Conjunction (NPIVM '99) with the 8th ACM Internation Conference on Information and Knowledge Management*, New York, NY, USA: ACM, 1999, p. 26–34.

GARCIA, R. E. *$ViDA_{ESE}$: processo de visualização exploratória para apoio a estudos experimentais em verificação, validação e teste de software*. Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação (ICMC-USP), 2006.

HANNEMANN, J. Aspect-Oriented Refactoring: Classification and Challenges. In: *Workshop on Linking Aspect Technology and Evolution (LATE '06) at 5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, 2006.

HORWITZ, S.; REPS, T.; BINKLEY, D. Interprocedural Slicing Using Dependence Graphs. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*, New York, NY, USA: ACM, 1988, p. 35–46.

HUANG, J.; LU, Y.; YANG, J. Aspect Mining Using Link Analysis. In: *Proceedings of the 5th International Conference on Frontier of Computer Science and Technology (FCST '10)*, Washington, DC, USA: IEEE Computer Society, 2010, p. 312–317.

HUMAN COMPUTER INTERACTION LAB TreeViz (TM) for Macintosh. Disponível em http://www.cs.umd.edu/hcil/pubs/treeviz.shtml, 2013.

JOHNSON, B.; SHNEIDERMAN, B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proceedings of the 2nd International IEEE Conference*

on Visualization (VIS '91), Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, p. 284–291.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of Test Information to Assist Fault Localization. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, New York, NY, USA: ACM, 2002, p. 467–477.

KATTI, A.; BINGI, V.; CHAVAN, V. Application of Program Slicing for Aspect Mining and Extraction - A Discussion. *International Journal of Computer Applications*, v. 38, n. 4, p. 12–15, 2012.

KEIM, D. A.; KRIEGEL, H.-P. Visualization Techniques for Mining Large Databases: A Comparison. *IEEE Transactions on Knowledge and Data Engineering*, v. 8, n. 6, p. 923–938, 1996.

KEIM, D. A.; PANSE, C.; SIPS, M. Information Visualization: Scope, Techniques and Opportunities for Geovisualization. In: DYKES, J.; MACEACHREN, A.; KRAAK, M.-J., eds. *Exploring Geovisualization*, Oxford: Elsevier Ltd., 2005.

KELLENS, A.; MENS, K.; TONELLA, P. A Survey of Automated Code-Level Aspect Mining Techniques. In: *Transactions on Aspect-Oriented Software Development (TAOSD)*, Berlin, Heidelberg: Springer-Verlag, p. 145–164, 2007.

KERIEVSKY, J. *Refactoring to Patterns*. Addison-Wesley Professional, 400 p., 2004.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; MARC LOINGTIER, J.; IRWIN, J. Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, Springer-Verlag, 1997, p. 220–242.

KRINKE, J.; BREU, S. Aspect Mining Based on Control-Flow. In: *Proceedings of the 7th Workshop Software Reengineering (WSR '05)*, Bad Honnef, Germany: GI-Softwaretechnik-Trends, 2005, p. 39–40.

KUCK, D. J.; KUHN, R. H.; PADUA, D. A.; LEASURE, B.; WOLFE, M. Dependence Graphs and Compiler Optimizations. In: *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL '81)*, New York, NY, USA: ACM, 1981, p. 207–218.

LAMPING, J.; RAO, R. The Hyperbolic Browser: A Focus+Context Technique for Visualizing Large Hierarchies. *Jornal of Visual Languages and Computing*, v. 7, n. 1, p. 33–55, 1996.

LAMPING, J.; RAO, R.; PIROLLI, P. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In: *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, p. 401–408.

LANZA, M. *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Tese de Doutoramento, University of Berne, 2003.

LANZA, M.; DUCASSE, S. Polymetric views – a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, v. 29, n. 9, p. 782–795, 2003.

LANZA, M.; DUCASSE, S.; GALL, H.; PINZGER, M. CodeCrawler: An Information Visualization Tool for Program Comprehension. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, New York, NY, USA: ACM Press, 2005, p. 672–673.

LOGOS RESEARCH SYSTEMS Logos Bible Software User Manual. Disponível em http://www.logos.com/, 1993.

MÄNTYLÄ, M. *Bad Smells in Software – a Taxonomy and an Empirical Study*. Dissertação de Mestrado, Helsinki University of Technology, Department of Computer Science and Engineering, 2003.

MÄNTYLÄ, M.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, v. 11, n. 3, p. 395–431, 2006.

MÄNTYLÄ, M.; VANHANEN, J.; LASSENIUS, C. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In: *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, Washington, DC, USA: IEEE Computer Society, 2003, p. 381–384.

MARIN, M.; DEURSEN, A. V.; MOONEN, L. Identifying Crosscutting Concerns Using Fan-In Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 17, n. 1, p. 3:1–3:37, 2007.

MARTINS, R. M. Teste Estrutural de Programas Orientados a Aspectos. Faculdade de Ciências e Tecnologia – Universidade Estadual Paulista "Júlio de Mesquita Filho". Relatório Técnico, 2007.

MASSICOTTE, P.; BADRI, L.; BADRI, M. Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs. *Journal of Object Technology*, v. 6, n. 1, p. 67–89, 2007.

MENS, K.; KELLENS, A.; KRINKE, J. Pitfalls in Aspect Mining. In: *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, Washington, DC, USA: IEEE Computer Society, 2008, p. 113–122.

MENS, T.; TOURWÉ, T. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, v. 30, n. 2, p. 126–139, 2004.

MORTENSEN, M. *Improving Software Maintainability Through Aspectualization*. Tese de Doutoramento, Colorado State University, 2009.

NEILL, C. J.; GILL, B. Refactoring Reusable Business Components. *IT Professional*, v. 5, n. 1, p. 33–38, 2003.

NORTH, C.; SHNEIDERMAN, B. Snap-together visualization: A user interface for coordinating visualizations via relational schemata. In: *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*, New York, NY, USA: ACM, 2000, p. 128–135.

NORTH, C. L. *A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization*. Tese de Doutoramento, University of Maryland, 2000.

OPDYKE, W. F. *Refactoring Object-Oriented Frameworks*. Computer science, University of Illinois at Urbana-Champaign, 1992.

OTTENSTEIN, K. J.; OTTENSTEIN, L. M. The Program Dependence Graph in a Software Development Environment. New York, NY, USA: ACM, 1984, p. 177–184.

PFEIFFER, J.-H.; GURD, J. R. Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs. In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, New York, NY, USA: ACM, 2006, p. 146–157.

PREFUSE Prefuse – Information Visualization Toolkit. Disponível em http://prefuse.org/, 2012.

RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, v. SE-11, n. 4, p. 367–375, 1985.

ROY, C. K.; CORDY, J. R.; KOSCHKE, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, v. 74, n. 7, p. 470–495, 2009.

SANT'ANNA, C. N.; GARCIA, A. F.; V. F. G. CHAVEZ, C.; DE LUCENA, C. J. P.; V. STAA, A. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

SCALE COMPILER GROUP Scale – A Scalable Compiler for Analytical Experiments. Department of Computer Science, University of Massachusetts, Amherst MA. 01003, USA. Disponível em https://zweb.cs.utexas.edu/svn/scale/branches/a02/scale/, 2005.

SHNEIDERMAN, B. Tree visualization with tree-maps: a 2-d space-filling appoach. *ACM Transactions on Graphics (TOG)*, v. 11, n. 1, p. 92–99, 1990.

STASKO, J.; DOMINGUE, J.; BROWN, M. H.; PRICE, B. A., eds. *Software Visualization: Programming as a Multimedia Experience*. Cambridge, MA: MIT Press, 1998.

TAIRAS, R.; GRAY, J.; BAXTER, I. Visualization of Clone Detection Results. In: *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA: ACM, 2006, p. 50–54.

THE ECLIPSE FOUNDATION AJDT: AspectJ Development Tools - The Visualiser. Disponível em http://www.eclipse.org/ajdt/visualiser/, 2012.

TIP, F. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, v. 3, n. 3, p. 121–189, 1995.

TONELLA, P.; CECCATO, M. Aspect Mining Through the Formal Concept Analysis of Execution Traces. In: *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, Washington, DC, USA: IEEE Computer Society, 2004, p. 112–121.

TOURWÉ, T.; MENS, K. Mining Aspectual Views Using Formal Concept Analysis. In: *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM '04)*, Washington, DC, USA: IEEE Computer Society, 2004, p. 97–106.

TOURWÉ, T.; MENS, T. Identifying Refactoring Opportunities Using Logic Meta Programming. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*, Washington, DC, USA: IEEE Computer Society, 2003, p. 91–100.

WEISER, M. Program Slicing. *IEEE Transactions on Software Engineering*, v. 10, n. 4, p. 352–357, 1984.

WU, Y.; QU, H.; ZHOU, H.; CHAN, M.-Y. Focus + Context Visualization with Animation. In: *Proceedings of the 1st Pacific Rim Conference on Advances in Image and Video Technology (PSIVT '06)*, Berlin, Heidelberg: Springer-Verlag, 2006, p. 1293–1302.

ZHANG, Y. *An Ontology-based Program Comprehension Model*. Computer science and software engineering, University Montreal, Quebec, Canada, 2007.