
**JOÃO LUCAS PEREIRA DE SANTANA**

# IMPLEMENTAÇÃO DE TÉCNICAS SEMÂNTICAS DE MINERAÇÃO PARA IDENTIFICAR INDÍCIOS DE INTERESSES TRANSVERSAIS EM SOFTWARES ORIENTADOS A OBJETOS

**LAVRAS - MG**
**2011**

**JOÃO LUCAS PEREIRA DE SANTANA**

# IMPLEMENTAÇÃO DE TÉCNICAS SEMÂNTICAS DE MINERAÇÃO PARA IDENTIFICAR INDÍCIOS DE INTERESSES TRANSVERSAIS EM SOFTWARES ORIENTADOS A OBJETOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador

Prof. Dr. Heitor Augustus Xavier Costa

**LAVRAS - MG**
**2011**

**JOÃO LUCAS PEREIRA DE SANTANA**

# IMPLEMENTAÇÃO DE TÉCNICAS SEMÂNTICAS DE MINERAÇÃO PARA IDENTIFICAR INDÍCIOS DE INTERESSES TRANSVERSAIS EM SOFTWARES ORIENTADOS A OBJETOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 14 de JUNHO de 2011

Dr. Antônio Maria Pereira de Resende UFLA

Dr. Dr. Ahmed Ali Abdalla Esmin UFLA

Dr. Heitor Augustus Xavier Costa

Orientador

**LAVRAS - MG**
**2011**

## AGRADECIMENTOS

A todos os professores do DCC – UFLA, que durante estes quatro anos guiaram meus passos na busca por conhecimento.

A meus amigos de curso, em especial aos da comp2007/01, que estiveram presentes nas horas de alegria e nas horas difíceis.

A minha família. Agradeço a todos vocês, que são os responsáveis por me fazer chegar aqui.

Em especial a minha mãe Joana. Que me deu apoio em todas as decisões que eu tomei. Que me aconselhou sempre que eu tive dúvidas. Que me deu suporte sempre que eu tive dificuldades.

**RESUMO**

A atividade de manutenção é realizada visando a alteração da estrutura do software garantindo sua integridade. Esta tarefa torna-se menos árdua quando se tem ampla compreensão do comportamento dos diversos componentes do software. Visando a obtenção desta compreensão, pesquisas surgiram propondo abordagens para identificar e separar códigos de uma funcionalidade específica que se encontram entrelaçados e/ou espalhados por vários módulos da aplicação, denominados interesses transversais. Considerando esses fatos, são apresentados neste trabalho o funcionamento de algumas técnicas de detecção de indícios de interesses transversais e uma comparação analisando critérios de granularidade dos resultados, envolvimento do usuário, sintomas da existência de interesses transversais e tipos de informações necessárias para computação das técnicas. Além disso, é abordado neste trabalho o desenvolvimento e comparação de três apoios computacionais em forma de *plug-ins* para o Eclipse visando a detecção de indícios de interesses transversais em software orientado a objetos escritos na linguagem de programação Java. A execução e a avaliação desses *plug-ins* foram realizadas em três softwares reais e os resultados são apresentados considerando critérios como cobertura e a relação entre tempo de execução e tamanho do software.

Palavras-chave**:** Manutenção de software. Interesses transversais. Técnicas semânticas de mineração. Orientação a Aspectos

**ABSTRACT**

Maintenance activity is performed aiming modify the software structure ensuring their integrity. And this task becomes easier when we have a full understanding of several system components behavior. Aiming to gain this understanding, researchs emerged suggesting approaches to identify and separate codes of a specific functionality which are entangled or scattered through various application modules, called crosscutting concerns. Considering these facts, are presented in this paper the behavior of certain techniques for crosscutting concerns indications detection and a comparision criteria considering results granularity, user involvement, symptoms of existence of crosscutting concerns and informations types required to technique computation. In addition, is discussed in this paper the development and comparison of three computational tools as plug-ins for Eclipse environment aiming to detect crosscutting concerns indications in Object-Oriented Software in Java. The execution and assessment of theses plug-ins were performed in three real software systems and the results are presented considering criteria as covering and relation between execution time and the software size.

Keywords***:*** Software maintence. Crosscutting concerns. Semantic mining techniques. Aspect orientated.

## LISTA DE FIGURAS

## LISTA DE TABELAS

**SUMÁRIO**

# 1. INTRODUÇÃO

Produtos de software são delineados e construídos por engenheiros de software e englobam desde programas executados em computadores de diversas escalas e arquiteturas a documentos em formato impresso ou digital associados a eles. Eles correspondem à entidade de software disponível para liberação a um usuário e passam por várias fases durante seu ciclo de vida (Definição, Desenvolvimento e Manutenção) (Sommerville, 2010).

Durante a fase Definição, devem-se identificar as informações a serem manipuladas, as funções a serem processadas, o nível de desempenho desejado, as interfaces a serem oferecidas, as restrições do projeto e os critérios de validação. Na fase Desenvolvimento, são abordadas questões de arquitetura do software, de estruturas de dados, sobre a forma como o projeto será codificado utilizando alguma linguagem de programação, dentre outros assuntos. A fase Manutenção é caracterizada pela realização de alterações das mais diversas naturezas, seja para corrigir erros residuais da fase anterior (manutenção corretiva), seja para incluir novas funções exigidas pelo cliente (manutenção evolutiva), seja para adaptar o software a novas configurações de hardware (manutenção adaptativa) (Pfleeger; Atlee, 2010; Sommerville, 2010; Pressman, 2009).

Durante a fase Manutenção, deve-se concentrar esforços para realizar alterações no software preservando sua integridade. No entanto, fatores de âmbito administrativo e/ou de implementação podem dificultar a execução desta tarefa, por exemplo, o software deve permanecer disponível enquanto as modificações são efetuadas, o tempo disponibilizado para os responsáveis

compreenderem o funcionamento do software pode ser curto e a documentação relacionada ao software pode ser deficitária ou mesmo inexistente.

Além dos problemas, percebe-se nos softwares atuais a presença de interesses transversais que correspondem a codificações relacionadas a uma determinada responsabilidade podem estar distribuídas em muitas partes do código fonte. Mesmo que um software seja bem projetado e decomposto em unidades modulares, algumas de suas funções atravessam módulos rompendo os princípios do encapsulamento (Ceccato *et al*., 2005; Tarr *et al*., 1999).

A dificuldade de garantir o encapsulamento é uma característica inerente da tecnologia de orientação a objetos (OO) (Bhatti; Ducasse, 2008). Uma forma de combater essa dificuldade, a qual não pode ser solucionada de maneira satisfatória utilizando técnicas tradicionais de programação (Cojocar; Serban, 2007), é utilizar a tecnologia de orientação a aspectos (OA) (Kiczales *et al*., 1997). As linguagens de programação orientadas a aspectos fornecem poderoso recurso o qual possibilita o encapsulamento de interesses adicionando um mecanismo extra de abstração, chamado Aspecto (Bruntink *et al*., 2005).

O sucesso da tecnologia de OA fez surgir questões relacionadas à possibilidade de transformar softwares orientados a objetos em orientados a aspectos (Tourwe; Mens, 2004). Este processo de transformação de software é chamado Refatoração Orientada a Aspectos, cujo intuito é modularizar os interesses transverais identificando-os e representando-os utilizando aspectos (Laddad, 2006) *apud* (Anbalagan; Xie, 2007). Essa refatoração tem duas etapas:

- *Aspect Mining*. Identificar interesses transversais (Yuen; Robillard, 2007) e/ou candidatos a aspectos (Tourwe; Mens, 2004) no código fonte;

- *Aspect Refactoring*. Definir aspectos apropriados e reestruturar o código base utilizando a tecnologia de orientação a aspectos (Tourwe; Mens, 2004) preservando o comportamento externo do software (Anbalagan; Xie, 2007).

É possível perceber que, enquanto *Aspect Mining* é a atividade de descoberta de interesses transversais que potencialmente podem ser transformados em aspectos, *Aspect Refactoring* é a atividade de transformar esses potenciais aspectos em aspectos reais no software (Kellens *et al*., 2007). Além disso, *Aspect Refactoring* se concentra nos desafios e no problema de determinar os pontos de execução do software a serem interceptados e redirecionados para o código do aspecto, visando sempre manter o comportamento original do software enquanto as funções transversais são modularizadas (Binkley *et al*., 2005).

Existem três distinções possíveis que podem ser consideradas na *Aspect Mining* (Kellens *et al*., 2007): i) identificação de interesses durante as fases iniciais do ciclo de desenvolvimento de software, conhecido como *Early Aspect* (Baniassad *et al*., 2006); ii) utilização de ferramentas para navegar pelo código fonte a procura de trechos com presença de interesses transversais, denominada *Aspect Browser* (Yoshikiyo *et al*., 1999); e iii) utilização de processos automatizados para detecção de sintomas de interesses transversais no código fonte. Os processos automatizados da terceira distinção utilizam técnicas de mineração e análise de dados, como princípios de clusterização, bem como outros procedimentos clássicos de análise de código para verificar a existência de indícios de interesses transversais, os quais, possivelmente, tornar-se-ão aspectos (Kellens *et al*., 2007).

## 1.1. Motivação

Diversos tipos de interesses transversais podem ser encontrados nos softwares existentes. Alguns mencionados com maior frequência na literatura são os interesses transversais de persistência, de gerenciamento de *logs*, de autenticação, de distribuição, de seguraça, de mobilidade e de tratamento de erros. Embora *Aspect Mining* seja uma área de pesquisa relativamente nova, diversas estratégias para identificação de interesses transversais foram propostas (Tourwe; Mens, 2004; Breu; Krinke, 2004; Zhang; Jacobsen, 2007).

Por causa do tamanho dos softwares, da complexidade de implementação e da ausência/desatualização da documentação, é necessário o apoio de ferramentas e de técnicas computacionais para ajudar os engenheiros de software a localizar/documentar os interesses transversais bem como ferramentas e metodologias para transformar em aspectos os interesses transversais descobertos (Kellens *et al*., 2007).

Um fator motivador desta pesquisa é a possibilidade de utilização da Refatoração Orientada a Aspectos para identificar claramente os interesses transversais no software visando a obtenção de melhor entendimento do código fonte e a facilidade para sua manutenção e sua evolução. A identificação de interesses transversais pode ser benéfica, pois destaca sutis oportunidades de refatoração no código fonte viabilizando o uso da Programação Orientada a Aspectos (POA) (Marin *et al*., 2004) (Shepherd *et al*. 2005).

## 1.2. Objetivo

A maioria dos métodos para detecção de interesses transversais está em um estágio prematuro e técnicas automatizadas ainda não estão bem fundamentadas (Qu; Liu, 2007). Além disso, a aplicação manual de técnicas de

mineração de interesses transversais em softwares legados é um processo difícil e propenso a erros (Kellens *et al*., 2007).

O objetivo desse trabalho é implementar três técnicas semânticas de mineração para identificar indícios de interesses transversais em software orientado a objeto escritos na linguagem de programação Java. Essa implementação são *plug-ins* para o ambiente de desenvolvimento Eclipse[1].

## 1.3. Metodologia

### 1.3.1. Tipo de Pesquisa

Analisando as diretrizes do método científico, esta pesquisa pode ser caracterizada em [Jung, 2004; Lakatos, Marconi 2001]:

- Natureza tecnológico-aplicada: pois utiliza conhecimentos existentes para criação de novos produtos;
- Objetivos de caráter descritivo: uma vez que tem a finalidade de observar, registrar fenômenos técnicos;
- Procedimentos experimentais em um trabalho de campo: pois são desenvolvidos protótipos de software para serem utilizados em estudos de caso visnado investigar fenômenos dentro de um contexto.
- Tempo de aplicação transversal: visto que os estudos de caso são realizados em um instante do tempo.

### 1.3.2. Procedimentos Metodológicos

Os estudos foram iniciados realizando um levantamento bibliográfico com uma pesquisa ampla na Internet, em livros, acervos digitais de artigos científicos e publicações relacionadas ao tema, por exemplo nas bibliotecas digitais da ACM e da IEEE. Posteriormente, foi estudado a tecnologia de OA,

[1] http://www.eclipse.org/

com foco na linguagem de programação AspectJ, de forma a verificar suas características, suas técnicas e suas contribuições para a área de computação.

A etapa seguinte consistiu da análise e da avaliação da aplicabilidade das principais técnicas de detecção de indícios de interesses transversais com propósito de refatorar softwares legados orientados a objetos. Posteriormente, o ambiente de desenvolvimento Eclipse e suas bibliotecas foram estudados e diversas abordagens para criação de *plug-ins* foram analisadas visando ao melhor conhecimento do ambiente de desenvolvimento e suas caracteríticas. Em seguida, três técnicas foram escolhidas para serem implementadas no Eclipse na forma de *plug-ins*. Após implementados, os *plug-ins* foram comparados com o objetivo de conhecer a relação entre tempo de análise e tamanho do software. Além disso, as implementações foram comparadas com relação à cobertura dos resultados encontrados.

Durante o andamento do trabalho, ocorreram reuniões periódicas entre o autor e seu orientador visando a manutenção da consistência, do foco e do bom andamento do trabalho.

## 1.4. Estrutura do Trabalho

Este trabalho está organizado da seguinte forma.

Breve histórico do surgimento da tecnologia de OA e seus conceitos básicos, considerando características da linguagem de programação AspectJ são apresentados no Capítulo 2.

Técnicas encontradas na literatura para identificar indícios de interesses transversais em softwares orientados a objetos são descritas no Capítulo 3. Além disso, os pré-requisitos são evidenciados e critérios de classificação são adotados visando a compação das abordagens de identificação de interesses transversais.

Ferramentas que auxiliaram no desenvolvimento dos apoios computacionais propostos neste trabalho, enfatizando o ambiente Eclipse, sua estrutura e componentes, são abordadas no Capítulo 4. Nesse capítulo, são tratadas algumas características de construção de *plug-ins* para o ambiente Eclipse.

Alguns trabalhos relacionados para identificação de interesses transversais em softwares orientados a objetos são apresentados resumidamente no Capítulo 5.

Estratégias e decisões tomadas durante o desenvolvimento deste trabalho são descritas no Capítulo 6. Além disso, é apresentado o desenvolvimento de três *plug-ins* para o Eclipse para identificação de interesses transversais em software orientado a objetos escritos na linguagem de programação Java.

Os *plug-ins* desenvolvidos correspondentes a três técnicas semânticas de mineração de interesses transversais em software orientado a objetos, foram utilizados em três softwares reais e os resultados obtidos são analisados no Capítulo 7.

Conclusões e contribuições dos *plug-ins* desenvolvidos e sugestões de trabalhos futuros são apresentados no Capítulo 8.

# 2. ORIENTAÇÃO A ASPECTOS

## 2.1. Considerações Iniciais

Principais conceitos da tecnologia de OA, cujo objetivo é tentar solucionar problemas não abordados pela tecnologia de OO, são apresentados nesse capítulo. A terminologia utilizada neste trabalho, quando referente à POA, segue a tradução utilizada pela Comunidade Brasileira de Desenvolvimento de Software Orientado a Aspectos (AOSDbr) para a Língua Portuguesa (AOSDbr, 2007).

Breve histórico do surgimento de POA e suas vantagens com relação à OO são apresentados na Seção 2.2. Conceitos básicos de OA, considerando os elementos da linguagem de programação AspectJ são descritos na Seção 2.3.

## 2.2. Surgimento da Programação Orientada a Aspectos

Softwares eram desenvolvidos utilizando programação desestruturada. Comandos de desvio de fluxo (por exemplo, `GOTO`) eram utilizados indiscriminadamente, tornando árduo o entendimento, a reutilização e a manutenção do código fonte. A programação estruturada surgiu como um grande avanço, pois funções/procedimentos eram agrupados e chamados a partir de um programa principal. Usuários da programação estruturada podiam reutilizar funções e procedimentos, desde que devidamente projetados (Resende; Silva, 2005). Nesta época, a reusabilidade foi explorada utilizando o conceito de bibliotecas.

O advento da tecnologia de OO, em meados da década de 70, marcou o início de uma nova etapa na história do desenvolvimento de software, na qual o reúso foi amplamente difundido e incentivado e as entidades do mundo real foram representadas por meio de objetos, classes, atributos e métodos. Nesta

tecnologia, cada classe torna-se responsável pela implementação de um interesse visando a garantia do encapsulamento das funções do software.

No entanto, garantir o encapsulamento é uma tarefa onerosa quando se dispõem apenas dos elementos de OO (Tarr *et al*., 1999), pois a existência de entrelaçamento do código das funções dificulta a decomposição em unidades modulares resultando no espalhamento da codificação da funcionalidade do software (Ceccato *et al*., 2005).

A *separação de interesses* (Kiczales *et al*., 1997) é um princípio da engenharia de software introduzido por Edsger Wybe Dijkstra (Dijkstra, 1997) e refere-se à habilidade de identificar, encapsular e manipular partes do software relevantes para um objetivo ou propósito particular (Cojocar; Serban, 2007). Uma forma de garantir a separação dos interesses espalhados pelo código fonte da aplicação é usufruir das características de OA (Tourwe; Mens, 2004). Pois, esta é uma tecnologia que possibilita o encapsulamento de interesses transversais, adicionando mecanismo extra de abstração, chamado Aspecto (Bruntink *et al*., 2005).

O expressão "Programação Orientada a Aspectos" surgiu em meados da década de 90 no Palo Alto *Reserach Center* (PARC). Naquele tempo, pesquisadores refletiam sobre *separação de interesses* e não estavam satisfeitos com as abordagens disponíveis para integrar interesses adicionais às principais funções dos softwares. Essa insatisfação foi causada por soluções existentes serem apenas extensões de linguagens de programação conhecidas providas de alguma estrutura dedicada a tratar do interesse. Por exemplo, a proposta de uma extensão da linguagem de programação C++ projetada para lidar com sistemas distribuídos denominada *Distributed Real-time Object Languagem* (DROL) (Takashio; Tokoro, 1992 *apud* Filman *et al*., 2004).

Naquela ocasião, os esforços eram para encontrar uma solução para garantir a separação de interesse nos níveis conceitual e de implementação (Filman *et al*., 2004). No nível conceitual, a separação de interesses precisa atentar para a questão de (1) fornecer uma abstração suficiente para cada interesse como um conceito individual e de (2) assegurar que os conceitos individuais sejam primitivos, no sentido de que eles abranjam um interesse natural na concepção do programador. No nível de implementação, a separação de interesse precisa fornecer uma adequada organização que isole os interesses uns dos outros. O objetivo deste nível é separar os blocos de código, cada um responsável por um interesse e com baixo acoplamento entre eles.

Neste modelo de separação, os interesses identificados no nível conceitual eram mapeados ao nível de implementação utilizando uma linguagem de programação. Algumas técnicas de programação prometiam tal mapeamento provendo mecanismos para interceptar envios e recebimentos de mensagens. Algumas delas são (Filman *et al*., 2004):

- *Meta-Level Programming*. As construções básicas das linguagens de programação, tais como classe e métodos, são descritas em um meta-nível e podem ser estendidas ou redefinidas por uma meta-programação. Cada objeto é associado a um meta-objeto, responsável pela semântica da operação no objeto base, utilizando um meta-*link*. O suporte a separação de interesses é garantido pela interceptação, por parte dos meta-objetos e de mensagens enviadas e recebidas pelos objetos. Os meta-objetos têm a oportunidade de executar ações em nome de um interesse de propósito especial;
- *Adaptative Programming*. É um modelo de programação baseado em padrões de código classificados em diferentes categorias visando a captura de abstrações durante a codificação, tais como, padrões de propagação, de

transporte e de sincronização. Cada categoria é atribuída a um interesse particular e pode ser vista como um componente de software que interage com outros componentes;

- *Composition Filters*. Estende a tecnologia de OO com a adição de filtros de composição de objetos, cujo propósito é gerenciar as mensagens enviadas e recebidas pelos objetos, acrescentar condições para aceitação e rejeição e determinar uma ação apropriada. A separação de interesses é alcançada definindo um filtro para cada interesse.

Como os recursos existentes não resolviam o problema de separação de interesses em sua totalidade, pesquisadores do PARC desenvolveram diversos projetos, tratando interesses específicos, em direção a uma solução de propósito geral. O projeto inicial foi denominado RG (Mendhekar *et al*., 1997), designado para aperfeiçoar o uso de memória na composição de funções contendo laços de repetição em cálculos de matrizes. Outro projeto estudado no centro de pesquisas utilizava o MatlLab para otimizar o utilização de memória. Annotated MatLab (Irwin *et al*., 1997) possuía diretivas no código fonte informando ao processador de código os pontos nos quais poderiam ser gerado código otimizado. Em seguida, entre os anos de 1995 e 1997, John Lamping trabalhou em um projeto chamado ETCML (Filman *et al*., 2004), o qual fornecia um conjunto de diretivas com o objetivo de instruir o processador da linguagem sobre quando avaliar certos trechos de código.

Uma abordagem mais aprimorada foi considerada por Cristina Lopes no projeto DJ (Lopes, 1997). Além de realizar uma análise *top-down* semelhante às ideias anteriores, a autora definiu construções de linguagem que permitiram que o código fosse reorganizado e certos interesses fossem "desembaraçados" de uma forma *bottom-up*. Posteriormente, Kiczales coordenou um projeto de implementação de um pré-processador (*weaver*) baseado no projeto DJ para uma

linguagem de programação chamada DJava, que deu origem a linguagem de programação AspectJ (Pawlak *et al*., 2005).

## 2.3. Elementos da Programação Orientada a Aspectos

Para compreender as características das linguagens programação orientadas a aspectos e conhecer as diretrizes que nortearam pesquisas na área, é necessário o entendimento de duas propriedades da tecnologia de OA (Filman *et al*., 2004):

- Inconsciência (*obliviousness*) assegura que não se pode dizer que o código do aspecto irá executar examinando o corpo do código base. Esta propriedade é desejável porque permite separação de interesses no processo de criação do software - interesses podem ser separados na concepção dos desenvolvedores e na estrutura do software;
- Quantificação (*quantification*) é o conceito relacionado a execução de um trecho de código em uma dada circunstância. Sobre esta propriedade, a POA direciona a utilização de instruções da seguinte forma: "Em um programa P, quando uma condição C ocorrer, execute a ação A". Isso implica em três escolhas para projetistas e desenvolvedores de softwares orientados a aspectos: i) quais tipos de condições podem ser especificadas; ii) como se dá a interação entre as ações e o programa; e iii) como mesclar a execução das ações e do programa.

A tecnologia de OA não é uma disciplina de programação que deve ser empregada isoladamente para desenvolvimento de software. Ao invés disso, essa tecnologia deve servir como complemento para facilitar a implementação, o entendimento, a manutenção e a evolução de softwares codificados em outros paradigmas. Para melhor entender os elementos abordados na tecnolgia OA,

segue a descrição de alguns termos específicos (Resende; Silva, 2005) (Filman *et al*., 2004) (Laddad, 2003):

- **Interesses (*Concerns*)**. Interesses são requisitos ou considerações específicas que devem ser tratadas para satisfazer o objetivo geral do software. Podem estar relacionados com implementações que variam desde questões de baixo nível (software deve armazenar valores em *cache*) a assuntos de alto nível (software deve ser de fácil usar - característica de usabilidade (ISO/IEC 9126, 2001));
- **Interesses Transversais (*Crosscutting Concerns*)**. Interesses transversais são implementações relacionadas a uma determinada responsabilidade no software distribuídas em muitas partes do código fonte (Marin *et al*., 2004; Roy *et al*., 2007). Em linhas gerais, são requisitos que demandam soluções espalhadas em diversos módulos do software. Este problema é conhecido como espalhamento de código (*Code Scattering*) (Marin *et al*., 2004; Roy *et al*., 2007). Um interesse transversal também pode ter sua codificação mesclada com códigos relacionados às principais funções de outros interesses da aplicação, problema denominado entrelaçamento de código (*Code Tangling*) (Ceccato *et al*., 2005; Marin *et al*., 2004). Exemplo de software com interesses espalhados no código fonte é apresentado na Figura 1. Nesse software, há o interesse de persistência, de *logging* e de segurança, os quais estão entrelaçados e espalhados em seus módulos.

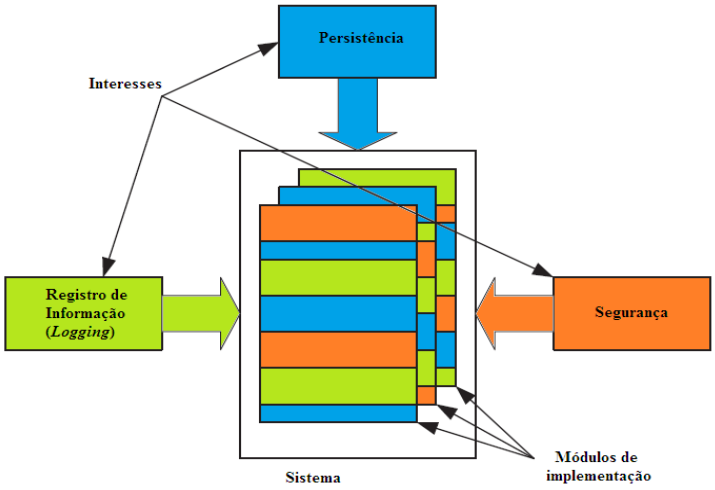


Figura 1 Interesses Transversais em um Software (Henry; Kafura, 1981)

- **Pontos de Junção (*Join Points*)**. Pontos de Junção são pontos bem definidos em uma estrutura ou no fluxo de execução de um software, nos quais um comportamento adicional pode ser anexado. Os elementos mais comuns de pontos de junção são chamadas de métodos, embora linguagens de programação orientadas a aspectos definam pontos de junção para outras circunstâncias (por exemplo, acesso, modificação e definição de atributos, exceções e execução de eventos). Se uma dessas linguagens permitir chamadas de métodos nos pontos de junção, um programador pode designar um código adicional para ser executado em uma chamada de método. Um exemplo dessa chamada de método é apresentado na Figura 2, quando um comportamento extra pode ser adicionado no momento da execução de um dado método;

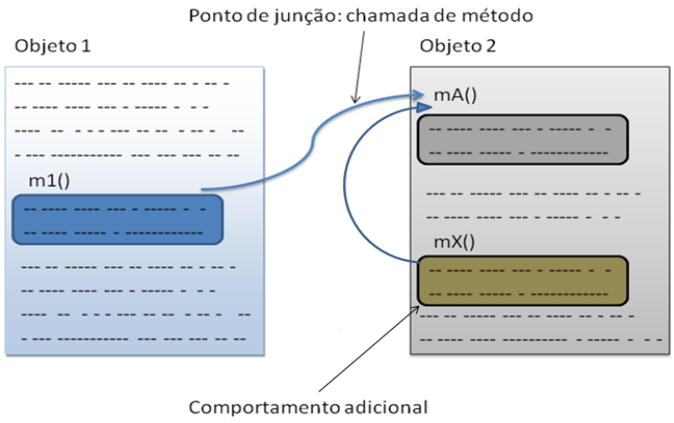


Figura 2 Ponto de Junção

- **Conjunto de Pontos de Junção (*Pointcut*)**. Conjunto de Pontos de Junção é uma estrutura que seleciona diversos *join points* e coleta o seu contexto. Ele é uma característica importante em linguagens de programação orientadas a aspectos, pois provêem a capacidade de atuação do aspecto em várias partes do software por meio de uma simples declaração. Este é o mecanismo conhecido como quantificação e é a maneira de executar trechos de códigos em vários pontos do software empregando uma simples instrução que identifica uma condição no fluxo de execução ou estrutura do software;
- **Adendo (*Advice*)**. Adendo é a ação executada quando um *join point* selecionado por um p*ointcut* é alcançado. No exemplo da Figura 2, a ação adicional é representada pele método *mX()* do Objeto 2. Um *advice* pode ser executado antes (*before*), depois (*after*) ou em substituição/a cerca (*around*) dos *join points* especificados. O mecanismo de inconsciência (*obliviousness*) é provido pelo *advice*, pois não há uma notação no *join point* informando que o código do *advice* será executado naquele ponto. Isso contrasta com as linguagens de programação convencionais, nas quais os mecanismos comuns de modularização de interesses (sub-rotinas) devem ser explicitamente chamados;

- **Introduções (*Introductions*)**. Introduções são instruções que inserem modificações nas classes, nas interfaces e nos aspectos do software. Por não alterarem diretamente o comportamento do software, elas são consideradas instruções entrecortantes estáticas. Como exemplo, métodos ou atributos podem ser inseridos em uma classe utilizando *introductions*;
- **Aspecto (*Aspect*)**. Aspecto é a unidade central da linguagem de programação AspectJ e contém o código que expressa a implementação de um interesse. Os *Pointcuts*, os *advices*, os *introductions*, os métodos e os atributos são combinados em um aspecto para determinar onde, como e quando determinada ação será executada;
- **Combinação (*weaving*)**. Combinação é o processo de composição das funções do núcleo do software com os aspectos, produzindo um software em sua forma final. O processo de *weaving* gerando um software a partir dos aspectos e componentes básicos do sistema é ilustrado na Figura 3.

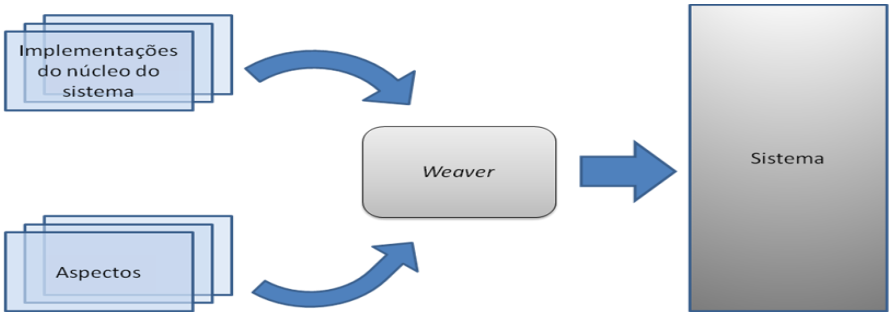


Figura 3 Processo de Combinação de Aspectos com Núcleo do Software

## 2.4. Considerações Finais

Neste capítulo, foram apresentados conceitos básicos e a gênese da tecnologia de OA. Pelo exposto, essa tecnologia procura resolver alguns problemas difíceis de serem tratados utilizando técnicas tradicionais de programação (por exemplo, tecnologia de OO). Para isso, ela proporciona a separação de interesses transversais presentes no código fonte em módulos

denominados Aspectos visando a diminuição do entrelaçamento e do espalhamento de código nos módulos do software.

# 3. TÉCNICAS SEMÂNTICAS DE MINERAÇÃO DE ASPECTOS

## 3.1. Considerações Iniciais

Algumas técnicas para detecção de interesses transversais são resumidamente apresentadas nesse capítulo, cuja essência consiste na busca por indícios de espalhamento e de entrelaçamento no código fonte de softwares orientados a objetos.

A mineração de aspectos pode ser conduzida de duas formas (Marin *et al*., 2004): i) abordagem *top-down*, a qual busca no código fonte por interesses transversais comumente tratados na literatura (como persistência e *log*); e ii) abordagem *bottom-up*, a qual analisa o código fonte em busca de sintomas da ausência de tratamento apropriado dos interesses transversais, isto é, espalhamento e entrelaçamento de código fonte. Apesar das distinções entre estas duas formas, o que ocorre na prática é uma combinação de ambas visando ao alcance melhores resultados.

A literatura relacionada a procedimentos para identificação de interesses transversais tem amadurecido e se desenvolvido bastante, o que contribui para o surgimento de novas técnicas. Algumas com grande variabilidade e distinção entre si, outras com diferenças sutis chegando ao ponto de se confundirem. Baseando-se em tais fatos, um conjunto de critérios para categorizar e comparar técnicas de identificação de interesses transversais foi proposto (Kellens *et al*., 2007):

- **Tipo de Dado de Entrada**. A análise pode ser feita a partir do código fonte do software (análise estática), a partir de informações geradas durante a execução do software (análise dinâmica) ou ambos;
- **Granularidade**. Algumas técnicas identificam aspectos em nível de métodos, outras fornecem granularidade mais refinada, informado

fragmentos de código ou declarações individuais como candidatos a aspectos;

- **Sintomas Analisados**. Uma vez que os principais indícios da existência de interesses transversais são o espalhamento e o entrelaçamento de código, as técnicas podem enfatizar a busca por um ou ambos;
- **Envolvimento do Usuário**. As técnicas podem diferir quanto ao grau de envolvimento do usuário durante o processo. Algumas carecem que o usuário conheça o software em estudo, outras necessitam apenas de filtragem dos resultados. Em resumo, técnicas menos automatizadas requerem maior intervenção do usuário.

O funcionamento da técnica que utiliza a métrica *Fan-In* para identificação de interesses transversais é descrita na Seção 3.2. As análises baseadas nos rastros de execução e no grafo de fluxo de controle são apresentadas na Seção 3.3 e na Seção 3.4, respectivamente. As análises da existência de clones como um indicativo de interesses transversais para encontrar candidatos a aspectos usando árvore sintática e grafo de dependência são apresentadas na Seção 3.5 e na Seção 3.6, respectivamente. A explicação da utilização de *clusters* e regras de associação no processo de mineração de aspectos é apresentada na Seção 3.7. Um quadro comparativo com algumas características das técnicas semânticas de mineração de aspectos em softwares orientados a objetos é apresentado na seção 3.8.

## 3.2. Análise *Fan-In*

### 3.2.1. Conceitos

A análise baseada na métrica *Fan-In* foi apresentada no 11º *Working Conference on Reverse Engineering* (WCRE) pelos pesquisadores Marius

Marin, Arie van Deursen e Leon Moonen, membros do *Software Evolution Research Lab*, situado na Holanda (Marin *et al*., 2004; Marin *et al*., 2007).

A análise *Fan-In* baseia-se na ideia que os principais sintomas da existência de interesses transversais em softwares orientados a objetos são causados por métodos invocados de diversas partes do código fonte e cuja funcionalidade é necessária em diferentes pacotes, classes e métodos do software. Tais métodos podem ser encontrados calculando a medida *Fan-In* a partir do grafo de chamadas estáticas do software.

O *Fan-In* de um método *A* é o número de fluxos locais no método *A*, somado ao número de estruturas de dados das quais o método *A* recupera informações (Henry; Kafura, 1981), sendo que existe um fluxo local de um método *A* para um método *B* se uma ou mais das seguintes condições são satisfeitas:

i) o método *A* chama o método *B*;

ii) se o método *B* chama o método *A* e o método *A* retorna o valor para o método *B*, o qual será posteriormente utilizado pelo método *B*;

iii) se o método *C* chama o método *A* e o método *B* passando o retorno do método *A* para o método *B*.

Um método relacionado a uma estrutura de dados *D* consiste dos procedimentos que diretamente atualizam ou recuperam informações de *D*. O *Fan-In* de um módulo é o número de locais pelos quais o controle é passado (chamadas para o método em análise) somado ao número de acessos a dados globais (Marin *et al*., 2004).

As informações necessárias para a computação da análise *Fan-In* são adquiridas estaticamente do código fonte do software. Vale ressaltar que a granularidade desta técnica está em nível de métodos e o principal sintoma da

existência de interesses transversais é o espalhamento de código. Além disso, o envolvimento do usuário ao final do processo é necessário para selecionar os candidatos a aspectos de uma lista de métodos ordenados com base no valor *Fan-In* calculado.

### 3.2.2. Explicação do Funcionamento

Para esclarecer o funcionamento desta técnica, seus desenvolvedores consideram o *Fan-In* de um método *m* como o número de métodos distintos que podem invocar *m* (Marin *et al*., 2004). Por causa do polimorfismo, chamadas a um método podem afetar o *Fan-In* de diversos outros métodos. Desta forma, invocações a um método *m* contribuem para o *Fan-In* de outros métodos implementados por *m* bem como os métodos que implementam *m*.

O Diagrama de Classes (UML, 2011) apresentado na Figura 4 ilustra o funcionamento da análise de *Fan-In* e a Tabela 1 apresenta valores *Fan-In* de cada método desse diagrama. O método `m()` da classe `A1` tem o seu valor *Fan-In* incrementado nas chamadas dos métodos `f1()`, `f3()`, `f4()` e `f5()` totalizando o valor 4, semelhante ao *Fan-In* do método `m()` da classe `A2`, incrementado nas chamadas dos métodos `f2()`, `f3()`, `f4()` e `f5()`. O método `m()` da classe `B` tem o seu valor *Fan-In* incrementado nas chamadas dos métodos `f1()`, `f2()`, `f3()`, `f4()` e `f5()` totalizando o valor 5. O método `m()` da classe `C1` tem o seu valor *Fan-In* incrementado nas chamadas dos métodos `f1()`, `f2()`, `f3()` e `f4()` totalizando o valor 4. O método `m()` da classe `C2` tem o seu valor *Fan-In* incrementado nas chamadas dos métodos `f1()`, `f2()`, `f3()` e `f4()` totalizando o valor 4.

Visando a obtenção de melhores resultados na utilização da técnica *Fan-In*, foi proposta uma sequência de passos a serem seguidos (Marin *et al*., 2004):

Passo 1. Computação automática da métrica *Fan-In* para os métodos no código fonte alvo. O resultado pode ser apresentado como uma lista ordenada pelo valor *Fan-In* e utilizado para inspecionar o local das chamadas de um método selecionado.

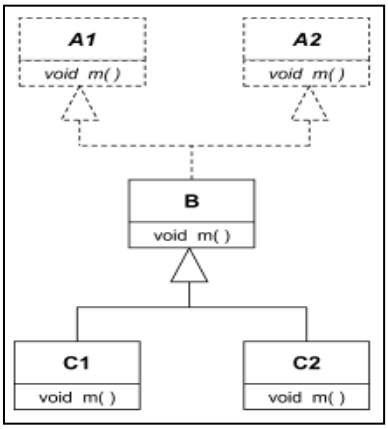


Figura 4 Diagrama de Classes (Vallée-Rai et al., 1999)

Tabela 1 Valores Fan-In para Cada Método (Vallée-Rai et al., 1999)

| | **Contribuições para o *Fan-In*** | | | | |
|---|---|---|---|---|---|
| **Locais de Chamada** | `A1.m` | `A2.m` | `B.m` | `C1.m` | `C2.m` |
| `f1(A1 a1){a1.m();}` | 1 | 0 | 1 | 1 | 1 |
| `f2(A2 a2){a2.m();}` | 0 | 1 | 1 | 1 | 1 |
| `f3(B b){b.m();}` | 1 | 1 | 1 | 1 | 1 |
| `f4(C1 c1){c1.m();}` | 1 | 1 | 1 | 1 | 0 |
| `f5(C2 c2){c2.m();}` | 1 | 1 | 1 | 0 | 1 |
| ***Fan-In* Total** | **4** | **4** | **5** | **4** | **4** |

Passo 2. Filtragem dos resultados do Passo 1.

a. Restringir o conjunto de invocadores àqueles que atingem um determinado valor limitante. Normalmente, 5% dos métodos alcançam valores de *Fan-In* superiores a 10.

b. Filtrar métodos acessores. Inicialmente, analisar a assinatura e, em seguida, analisar a implementação do método.

c. Filtrar do conjunto restante os métodos utilitários, tais como toString() e gerenciadores de coleções.

Passo 3. Análise manual dos resultados. Os elementos considerados são os invocadores e os locais de chamada, nomes e implementações dos métodos e comentários no código fonte.

Pode-se distinguir três situações em que um alto valor de *Fan-In* indica a presença de interesses transversais: i) quando o método é o elemento chave da codificação do interesse transversal, tais como métodos utilizados em classes de *debug*, *tracing* ou *logging*; ii) quando a implementação do interesse transversal está baseada em alguma função usual do software, espalhando-se ele, e o método com alto *Fan-In* participa desta implementação; e iii) quando são utilizados *Design Patters* (padrões de projeto) com estrutura distribuída pelo software, por exemplo, o padrão de projeto *Observer*.

## 3.3. Análise dos Rastros de Execução

### 3.3.1. Conceitos

A primeira abordagem para detectar interesses transversais em softwares legados utilizando informações geradas durante a execução do software foi proposta por Silvia Breu e Jens Krinke (Breu; Krinke, 2004) na 19º Int*ernational Conference on Automated Software Engineering*, em Linz, Áustria.

Os rastros de execução são informações sobre a sequência de chamadas de métodos produzidas durante a execução do software, as quais refletem o comportamento em tempo de execução do software (Kellens *et al*., 2007). Este método de investigação de interesses transversais analisa os dados coletados a procura de padrões de execução recorrentes e impõe restrições para definir quando uma amostra é recorrente. Um exemplo de restrição é a garantia de um

padrão existir em diferentes contextos de chamadas nos rastros de execução. Os padrões extraídos descrevem aspectos comportamentais do software e, quando implementam funções recorrentes no software, podem ser considerados interesses transversais (Breu; Krinke, 2004).

A granularidade de código considerada é o método, os subsídios para a computação desta técnica são gerados pela execução do software, enquadrando-a como uma técnica de análise dinâmica, o indicador da presença de interesses transversais é o espalhamento do código e o envolvimento do usuário durante processo é baixo, há a necessidade apenas de inspeção dos padrões recorrentes (Kellens *et al*., 2007).

### 3.3.2. Explicação do Funcionamento

Para que seja possível encontrar padrões recorrentes nos rastros de execução, torna-se necessário defini-los quanto à forma e estabelecer alguns conceitos. Como a tecnologia de OO encapsula os comportamentos em métodos com funções específicas, o primeiro conceito relacionado a rastros de execução é definido a partir das assinaturas dos métodos.

Um rastro de execução $T_P$ de um programa $P$ com assinaturas de métodos $\mathcal{N}_P$ é definido como uma lista $[t_i, \dots, t_n]$ de pares $t_i \in (\mathcal{N}_p \times \{ent, ext\})$, sendo $ent$ a entrada de um método e $ext$ a saída de um método. Intuitivamente, um rastro de execução é uma sequência de invocações e saídas dos métodos de um software.

Para facilitar a leitura e o entendimento de um rastro de execução, os símbolos $ent$ e $ext$ são substituídos por "{" e "}", respectivamente. Além disso, é imposta uma estrutura ao rastro de execução para que a assinatura do método $\mathcal{N}_P$ seja descartada dos pontos de saída de métodos, o que elimina informações

redundantes. Um exemplo de rastro de execução que assegura essa estrutura é apresentado na Figura 5. Adotada esta estrutura, os interesses transversais são demarcados por duas diferentes relações de execução possíveis de serem encontradas nos traços de execução.

```
 1 B() {          17   J() {}        33   }
 2   C() {        18 }               34 }
 3     G() {}     19 F() {           35 D() {
 4     H() {}     20   K() {}        36   C() {}
 5   }            21   I() {}        37   A() {}
 6 }              22 }               38   B() {
 7 A() {}         23 J() {}          39     C() {}
 8 B() {          24 G() {}          40   }
 9   C() {}       25 H() {}          41   K() {}
10 }              26 A() {}          42   I() {
11 A() {}         27 B() {           43     J() {}
12 B() {          28   C() {}       44   }
13   C() {        29   G() {}       45   G() {}
14     G() {}     30   F() {         46   E() {}
15     H() {}     31     K() {}      47 }
16   }            32     I() {}
```

Figura 5 Rastro de Execução (Marin et al., 2004)

Um método pode ser executado após a execução de um método antecedente. Por exemplo, na linha 12, o método `B()` é executado após o método `A()`, na linha 11. Esta é denominada uma relação de execução externa ou *outside-execution relation*. Outra relação de execução possível é na ocasião de um método ser chamado dentro do corpo de outro método, como exemplificado nas linhas 12 e 13. O método `C()` é invocado dentro do método `B()`. Esta é uma relação de execução interna ou *inside-execution relation* (Breu; Krinke, 2004). No entanto, esta notação não fornece as informações suficientes para o processo de minreação de aspectos. Por exemplo, os métodos `C()`, `A()` e `B()` são invocados, respectivamente, nas linhas 36, 37 e 38 pelo método `D()` na linha 35, mas não há informações que garantam a sequência das chamadas dos métodos. Para solucionar este problema, é necessária uma formalização da ordem das invocações.

Sejam $u, v \in \mathcal{N}_P$, $u \rightharpoonup v$ é chamada uma relação de execução externa-anterior se $[(u, ext), (v, ent)]$ é uma sublista de $T_P$. O conjunto das relações de

execução externa-anterior é denotado por $S^{\rightarrow}(T_P)$. Esta relação pode ter a ordem invertida, isto é, $v \leftarrow u$ é uma relação de execução externa-posterior se $u \rightharpoonup v \in S^{\rightarrow}(T_P)$ e o conjunto com os elementos deste tipo é expresso por $S^{\leftarrow}(T_P)$.

Sejam $u, v \in \mathcal{N}_P$, a relação $u \in_{\top} v$ é chamada uma relação de execução interna-inicial se $[(v, ent), (u, ent)]$ é uma sublista de $T_P$ e $u \in_{\perp} v$ é uma relação de execução interna-final se $[(u, ext), (v, ext)][(u, ext), (v, ext)]$ é uma sublista de $T_P$. Os respectivos conjuntos dessas relações são $S^{\in_{\top}}(T_P)$ e $S^{\in_{\perp}}(T_P)$.

Para o exemplo de rastro de execução da Figura 5, tem-se o conjunto $S^{\rightarrow}$ de relações de execução externa-anterior apresentado na Figura 6. O conjunto $S^{\leftarrow}$ das relações de execução externa-posterior foi suprimido, uma vez que é possível encontrá-lo invertendo a ordem das relações do conjunto $S^{\rightarrow}$.

$$
\begin{aligned}
S^{\rightarrow} = \{ & \mathtt{B()} \rightharpoonup \mathtt{A()}, \mathtt{G()} \rightharpoonup \mathtt{H()}, \mathtt{A()} \rightharpoonup \mathtt{B()}, \mathtt{C()} \rightharpoonup \mathtt{J()}, \\
& \mathtt{B()} \rightharpoonup \mathtt{F()}, \mathtt{K()} \rightharpoonup \mathtt{I()}, \mathtt{F()} \rightharpoonup \mathtt{J()}, \mathtt{J()} \rightharpoonup \mathtt{G()}, \\
& \mathtt{H()} \rightharpoonup \mathtt{A()}, \mathtt{B()} \rightharpoonup \mathtt{D()}, \mathtt{C()} \rightharpoonup \mathtt{G()}, \mathtt{G()} \rightharpoonup \mathtt{F()}, \\
& \mathtt{C()} \rightharpoonup \mathtt{A()}, \mathtt{B()} \rightharpoonup \mathtt{K()}, \mathtt{I()} \rightharpoonup \mathtt{G()}, \mathtt{G()} \rightharpoonup \mathtt{E()} \}
\end{aligned}
$$

Figura 6 Relações de Execução Externa-Anterior (Marin et al., 2004)

Os conjuntos das relações de execução interna-inicial ($S^{\in_{\top}}$) e interna-final ($S^{\in_{\perp}}$) são apresentados na Figura 7.

$$
\begin{aligned}
S^{\in_{\top}} = \{ & \mathtt{C()} \in_{\top} \mathtt{B()}, \mathtt{G()} \in_{\top} \mathtt{C()}, \mathtt{K()} \in_{\top} \mathtt{F()}, \mathtt{C()} \in_{\top} \mathtt{D()}, \\
& \mathtt{J()} \in_{\top} \mathtt{I()} \} \\
S^{\in_{\perp}} = \{ & \mathtt{H()} \in_{\perp} \mathtt{C()}, \mathtt{C()} \in_{\perp} \mathtt{B()}, \mathtt{J()} \in_{\perp} \mathtt{B()}, \mathtt{I()} \in_{\perp} \mathtt{F()}, \\
& \mathtt{F()} \in_{\perp} \mathtt{B()}, \mathtt{J()} \in_{\perp} \mathtt{I()}, \mathtt{E()} \in_{\perp} \mathtt{D()} \}
\end{aligned}
$$

Figura 7 Relações de Execução Interna-Inicial e Interna-Final (Breu; Krinke, 2004)

As relações de execução recorrentes podem ser vistas como indicadores dos mais gerais padrões de execução (Breu; Krinke, 2004). Mas, para determinar

quais relações de execução são recorrentes e, consequentemente, potenciais interesses transversais do sistema, torna-se essencial definir algumas restrições.

Por formalização técnica, é necessário declarar que não existem execuções de outros métodos entre a entrada e a saída de um método ou entre métodos aninhados. Tomando como apoio o rastro de execução apresentado na Figura 5 para exemplificar esta situação, pode-se ver que não há invocações dentro do método `A()` na linha 7. Além disso, não existem invocações de outros métodos dentro do método `B()` antes do método `C()` ser chamado nas linhas 12 e 13, respectivamente. Ou seja, não há chamadas de métodos a menos que estejam declaradas no rastro de execução do programa $T_P$. Esta ausência de invocação é representada pela assinatura de método nulo $\varepsilon$. O rastro execução do programa continua definido como antes, mas, a partir deste ponto, foram incluídas as assinaturas dos métodos contidas em $\mathcal{N}_P \cup \{\varepsilon\}$. Da mesma forma, os conjuntos $S^{\rightarrow}$, $S^{\leftarrow}$, $S^{\in_\top}$ e $S^{\in_\bot}$ também incluem as relações de execução envolvendo $\varepsilon$.

Formalmente, uma relação de execução $s = u \circ v \in S^{\circ}$, $\circ \in \{\rightarrow, \leftarrow, \in_\top, \in_\bot\}$ é chamada uniforme se $\forall w \circ v \in S^{\circ}: u = w, u, v, w \in \mathcal{N}_P \cup \{\varepsilon\}$. Isto é, existe sempre na mesma composição. O conjunto $\hat{U}^{\circ}$ contém as relações $s \in S^{\circ}$ que satisfazem esta restrição. Uma relação de execução $s = u \circ v \in U^{\circ} = \hat{U}^{\circ} \backslash \{u \circ v \mid u = \varepsilon \vee v = \varepsilon\}$ é chamada transversal se $\exists s' = u \circ w \in U^{\circ}: w \neq v, u, v, w \in \mathcal{N}_P$. Isto é, a relação $s$ ocorre em mais de um único contexto de chamada no rastro de execução do programa $T_P$. Para relações de execução internas ($u \in_\top v$ ou $u \in_\bot v$), o contexto da chamada é o método circundante $v$. Para relação de execução externas $u \rightarrow v$ ou $u \leftarrow v$, o contexto da chamada é o método $v$ invocado depois (ou antes) de um método $u$ a ser executado. $R^{\circ}$ é o conjunto de relações de execução $s \in U^{\circ}$ as quais

satisfazem este requisito. As relações de execução $s \in R^{\circ}$ são chamadas de candidatas a aspectos, uma vez que elas representam um potencial interesse transversal do software em análise.

Em resumo, a técnica de análise dos rastros de execução avalia o comportamento em tempo de execução do software classificando as relações de chamadas entre os métodos como externa-anterior, externa-anterior, interna-inicial e interna-final. Baseando-se nessas categorias, são definidos os conceitos de uniformidade e de transversalidade visando formalizar o significado de interesses transversal.

## 3.4. Análise Baseada no Grafo de Fluxo de Controle

### 3.4.1. Conceitos

Esta técnica semântica de detecção de interesses transversais complementa a abordagem de Análise dos Rastros de Execução (Breu; Krinke, 2004) descrita na Seção 3.3 com informações estáticas sobre o software em análise, bem como utiliza as definições fornecidas anteriormente. Essa técnica foi igualmente proposta por Jens Krinke e Silvia Breu (Krinke; Breu, 2004) e submetida ao *1º Workshop on Aspect Reverse Engineering*. Essa técnica leva em consideração os padrões recorrentes no código fonte baseando-se em diversas restrições (por exemplo, um determinado padrão ocorrer em diferentes contextos de chamada) para determinar a existência de possíveis interesses transversais no software.

A análise dinâmica considerada em várias abordagens de mineração de aspectos é uma métrica razoável para avaliar o comportamento em tempo de execução do software, pois examina as informações extraídas das chamadas de métodos ocorridas durante a sua utilização efetiva. No entanto, ao examinar essas informações, pode-se encontrar inconsistência nos dados coletados. Por

exemplo, a situação na qual uma estrutura de controle de fluxo redireciona a execução do software em duas direções distintas (Qu; Liu, 2007). Caso uma determinada condição seja verdadeira, um método `A()` é executado; caso contrário, o método `B()` é executado. Como o fluxo de execução é bifurcado, torna-se impossível avaliar a relação existente entre esses métodos, pois eles não se enquadram nas restrições proposta na técnica, mesmo estando presentes em locais adjacentes no código fonte. Assim, os resultados das análises estática e dinâmica são diferentes em decorrência de diversos fatores (Breu; Krinke, 2004).

O tipo de dado de entrada para computação desta técnica é definido como estático, pois apenas o código fonte do software é analisado. As demais características seguem as definidas para a técnica de Análise dos Rastros de Execução. O nível de granularidade é limitado a métodos, o espalhamento de código é o principal sintoma de interesse transversal e o envolvimento do usuário é baixo, sendo necessário apenas para vistoriar os padrões recorrentes encontrados ao final do processo.

### 3.4.2. Explicação do Funcionamento

As informações utilizadas na computação desta análise são extraídas do grafo de fluxo de controle do software em análise e a busca é feita diretamente por relações de execução uniforme e transversal, não se verificam as relações iniciais não sujeitas a restrições.

Com o objetivo de concentrar esforços no processo de detecção de interesses transversais, foi utilizado o apoio de uma ferramenta computacional, chamada Soot (Vallée-Rai *et al.*, 1999), para extrair do código fonte o grafo de fluxo de controle. Os resultados produzidos pelo Soot são analisados e o grafo é percorrido buscando relações de execução uniforme e transversal. O processo

realizado por esta abordagem parte do pré-suposto que o grafo do código fonte foi criado e está de acordo com as suas necessidades. Após esta etapa, as relações de execução são examinadas a fim de encontrar os padrões recorrentes. Como exemplo:

- Para relações de execução do tipo interna-inicial ($R^{\in_\tau}$), é extraída no grafo de fluxo de controle a chamada de método $u$ imediatamente seguinte à entrada de um método invocado $v$. Tal relação é uniforme se todo caminho para o método $u$ inicia-se com a mesma chamada de método $v$. Um determinado método $u$ é dito transversal se há no mínimo duas relações de execução uniforme $u \in_\tau v$ e $u \in_\tau w$, com $v \neq w$;
- Para as relações de execução externa-anterior ($R^{\rightharpoonup}$), extrai-se todo par de chamadas de método $u$, $v$ se existe um caminho da invocação do método $u$ para a invocação do método $v$ sem outras chamadas de método entre elas. Tal par é uma relação de execução externa-anterior uniforme $u \rightharpoonup v$ se os caminhos, partindo da invocação do método $u$, contêm o método $v$ como o próximo a ser chamado. O caso de uma relação ter comportamento transversal incide na mesma situação para a relação do tipo interna-inicial.

Em estudos de casos realizados em softwares de *benchmarks*, parte dos candidatos interesses transversais encontrados não puderam ser refatorados em aspectos. No entanto, percebeu-se melhor visão sobre o comportamento transversal do software em análise (Krinke; Breu, 2004).

### 3.5. Análise para Detecção de Clones

O problema básico da detecção de clones é descobrir se os fragmentos de código fonte produzem o mesmo resultado e, para facilitar esta detecção, deve-se dividir esse código em trechos e compará-los para determinar se os pares de fragmentos são equivalentes. Uma possibilidade prática e eficiente é

comparar representações do software nas quais os fluxos de controle e de dados estejam explícitos (Baxter *et al*., 1998).

As técnicas de detecção de clones tentam identificar trechos de código duplicados ou que tenham sofrido pequenas alterações a partir de um fragmento de origem. Diversas técnicas são descritas na literatura, diferenciando-se com relação à abstração do código fonte que cada uma emprega como subsídio para computação. Possíveis abordagens de detecção são (Bruntink, 2004):

- **Baseadas na AST**. A AST (*Abstract Sintax Tree* - Árvore Sintática Abstrata) é criada a partir do código fonte e a detecção de clones procura por subárvores similares;
- **Baseadas no PDG**. O PDG (*Program Dependence Graph* - Grafo de Dependência do Programa) é analisado para procurar subgrafos semelhantes, pois contém informações sobre fluxos de controle e dados.

### 3.5.1. Utilizando Árvore Sintática Abstrata

#### 3.5.1.1. Conceitos

A ideia de utilizar a AST para detecção de clones (Baxter *et al*., 1998) visando a identificação de interesses transversais foi proposta por Magiel Bruntink, Arie van Deursen, Tom Tourwe e Remco van Engelen (Bruntink *et al*., 2004) e submetida à 20ª *IEEE International Conference on Software Maintenance*. Posteriormente, Magiel Bruntink aperfeiçoou esta ideia utilizando métricas distintas para encontrar candidatos favoráveis a aspectos (Bruntink, 2004).

Durante o processo de detecção de clones, três principais algoritmos são aplicados sobre a AST gerada a partir do código fonte (Baxter *et al*., 1998): i) algoritmo para detectar subárvores clones; ii) algoritmo para detectar sequências de tamanho variável nas subárvores clones; e iii) algoritmo para procurar por

clones não idênticos tentando generalizar combinações de outros clones. Uma vez que os clones estão identificados, inicia-se sua análise para encontrar indícios de identificar interesses transversais. Níveis de classificação baseados em métricas podem ser utilizados com o objetivo de filtrar os clones mantendo aqueles que apresentam valores significativos para as métricas (Bruntink, 2004).

A granularidade dos resultados produzidos está em nível de fragmentos de código, o que fornece um retorno mais representativo ao usuário, podendo inclusive utilizar as informações obtidas para definir as estruturas dos *advices*. O tipo de dado de entrada é considerado estático. O espalhamento de código é o sintoma de interesse transversal avaliado e o envolvimento do usuário consiste na navegação e inspeção de trechos de códigos considerados clones.

#### 3.5.1.2. Explicação do Funcionamento

Após a identificação, os clones são agrupados e denominados *Classes Clones* (Kamiya *et al*., 2002 *apud* Bruntink *et al*., 2004). Para satisfazer questões formais, uma relação *clone* deve assegurar as propriedades de equivalência: i) reflexividade (um fragmento de código é um clone exato de si próprio); ii) simetria (se um código *A* é um clone exato de um código *B*, então o código *B* é um clone exato do código *A*); e iii) transitividade (se um código *A* é um clone exato de um código *B* e esse código *B* é um clone exato de um código *C*, então o código *A* é um clone exato do código *C*).

Depois de criadas as *Classes Clones*, filtros devem ser aplicados com o objetivo de limitar a quantidade de resultados apresentados, os quais podem ser baseados em métricas que considerem o rigor da duplicação de código para cada *Classe Clone* (Bruntink, 2004). Para uma dada *Classe Clone C*, as seguintes métricas podem ser avaliadas, sendo que valores mais altos correspondem a casos mais severos de duplicação de código:

- Número de Clones (NC) inclusos na *Classe Clone C*;
- Número de Linhas (NL) distintas na *Classe Clone C*;
- Tamanho Médio do Clone (TMC). Essa métrica corresponde ao NL dividido pelo NC.

Instâncias triviais de duplicação de código podem ser solucionadas utilizando técnicas tradicionais de reengenharia. Para outras, relacionadas a espalhamento de código, torna-se necessário a utilização de medidas que estejam de acordo com as necessidades da tecnologia de OA. As seguintes métricas capturam esta noção de espalhamento para uma dada *Classe Clone C*, sendo que valores mais altos indicam teores elevados de espalhamento:

- Número de Arquivos (NA) distintos nos quais a *Classe Clone C* ocorre;
- Número de Funções/Métodos (NFM) nos quais a *Classe Clone C* ocorre.

As *Classes Clones* devem estar ordenadas por algum critério, por exemplo seus valores de NL e NFM que indicam o grau de manutenibilidade ao aplicar o processo de refatoração de OA. Nesse caso, *Classes Clones* com valores altos de NL e NFM terão maior prioridade no processo de refatoração. Experimentos indicam eficiência em torno de 80% na identificação de interesses transversais relacionados a tratamento de erros, rastreabilidade, verificação de parâmetros e manipulação de memória [Bruntink *et al*., 2004].

### 3.5.2. Utilizando Grafos de Dependência

#### 3.5.2.1. Conceitos

A utilização de grafo de dependência do programa foi proposta por David Shepherd, Emily Gibson e Lori Pollock (Shepherd *et al*., 2004) na *International Conference on Software Engineering Research and Practice* como um aperfeiçoamento da análise da árvore sintática abstrata. Esta análise

considera a existência de clones como indicador da presença de interesses transversais no software. No entanto, os subsídios para a computação são fornecidos pelo grafo de dependência do software.

Diferentes definições da representação de dependência do software têm sido dadas, estando sempre vinculadas à intenção do seu uso (Horwitz; Reps, 1992). Entretanto, essas definições são variações do tema introduzido por David J. Kuck, Yoichi Muraoka e Shyh-Ching Chen (Kuck *et al*., 1972) e compartilham a característica comum de fornecer uma explícita representação das dependências de controle e dados. Na representação do PDG de um método, cada nó representa uma declaração e as arestas entre os nós representam as dependências de controle ou dados entre as declarações correspondentes (Shepherd *et al*., 2004).

Com o objetivo de identificar os clones, a técnica de análise do grafo de dependência utiliza a abordagem proposta por Raghavan Komondoor e Susan Horwitz (Komondoor; Horwitz, 2001), a qual particiona as declarações do software em classes equivalentes baseando-se na estrutura sintática, ignorando nomes de variáveis e valores de literais, removendo redundâncias e agrupando resultados.

Desta forma, um par de clones é identificado por subgrafos isomórficos do PDG. Outra abordagem de detecção de clones considerada na técnica de análise do grafo de dependência é a relatada por Jens Krinke (Krinke, 2001). Uma especialização do PDG é criada contendo os nós para expressões, variáveis, métodos e declarações. Os nós são ligados com arestas informando a ordem de avaliação, dependência entre os dados e valores armazenados nas variáveis.

Baseando-se na eficiência em identificar clones das técnicas descritas anteriormente, refinamentos foram feitos na técnica de análise do grafo de dependência visando a obtenção de benefícios para mineração de aspectos de forma automática. O foco desta abordagem é encontrar trechos que possam ser refatorados em aspectos com *advice* do tipo *before*.

A classificação da técnica apresentada nesta seção é equivalente àquela apresentada na Seção 3.5, o que difere é apenas no modo como os dados (estáticos) de entrada são computados. Para esta abordagem, o código fonte é analisado e representado como grafo de dependência, a anterior cria uma árvore sintática. A granularidade é em nível de fragmentos de código, o indicador de interesse transversal é o espalhamento de código e o envolvimento do usuário é necessário ao final do processo para inspeção dos resultados.

#### 3.5.2.2. Explicação do Funcionamento

O procedimento de mineração de aspectos abordado nesta seção está decomposto em quatro fases descritas a seguir. Desta maneira, possibilita-se melhor compreensão de cada etapa bem como realizar possíveis otimizações nas fases separadamente.

A primeira etapa é a transformação de um PDG em nível de representação intermediária (Komondoor; Horwitz, 2001; Krinke, 2001) para um PDG em nível de código, para que o usuário da técnica tenha melhor entendimento dos resultados. Nesta transformação, os nós do grafo intermediário que correspondem à mesma linha do código fonte são agrupados em um único nó. Para cada aresta que origina ou termina em um nó do PDG em nível intermediário, a aresta é incluída no PDG em nível de código no nó apropriado, evitando a inserção de arestas duplicadas no PDG em nível de código.

A segunda fase pode ser vista como um processo de detecção de clones e identificação do conjunto de candidatos a refatoração. Sabendo que o problema de determinar se dois subgrafos são isomórficos é NP-Completo (Krinke, 2001 *apud* Shepherd *et al*., 2004), torna-se necessário projetar o algoritmo de modo a escolher entre a acurácia da detecção de clones ou o desempenho da execução. Para conseguir o melhor aproveitamento deste *trade-off*, o algoritmo realiza as comparações entre as declarações utilizando a árvore sintática abstrata e o grafo de dependência. Dados dois PDGs representando diferentes métodos, o algoritmo inicia-se pelo nó de entrada e executa uma busca em largura em cada PDG realizando "podas" à medida que percorre o grafo. Os filhos do nó de entrada de um dos PDGs são comparados com cada um dos filhos do nó de entrada do outro PDG. Quando encontrados nós que sejam equivalentes de acordo com a comparação das declarações na AST, eles são armazenados em uma lista a ser analisada posteriormente. Após a comparação entre os nós de entrada, um nó é retirado da lista e um procedimento análogo é executado. O número de estados criados é bastante reduzido por causa da rigorosa comparação baseada na AST no momento da "poda". Além disso, a complexidade do processo de identificação é amortizada, pois o início dá-se apenas nos nós de entrada do PDG e não em qualquer ponto do método (Shepherd *et al*., 2004). Como os PDGs são comparados várias vezes durante o procedimento de identificação de clones, esta deve ser feita de maneira eficiente. Além disso, a comparação das AST é apenas estrutural; caso um nó seja diferente de outro nó na AST correspondente, as árvores são consideradas diferentes. Variáveis do mesmo tipo são consideradas equivalentes, assim como variáveis em uma hierarquia comum ou que uma seja subtipo da outra (Shepherd *et al*., 2004).

A terceira fase consiste na filtragem dos resultados indesejados. A dependência entre os dados pode ser usada para desconsiderar candidatos que dependam de algum nó que não faz parte de nenhum clone.

A quarta fase recebe das etapas anteriores um conjunto de pares de métodos e os agrupam em cojuntos maiores de candidatos similares. Este agrupamento produz resultados mais adequados para identificar os locais de espalhamento e duplicação de código e, possivelmente, adaptar o trecho de código de uma forma adequada à tecnologia de OA.

A eficiência deste método de detecção de Interesses Transversais é comprovada com a realização de testes com softwares reais e a obtenção de resultados expressivos, ou seja, cerca de 90% dos candidatos relatados foram considerados passíveis de refatoração.

## 3.6. Análise Utilizando *Clusters* e Regras de Associação

### 3.6.1. Conceitos

Este procedimento de combinação de *clusters* e regras de associação para análise de código foi sugerido por Lili He e Hongtao Bai (He; Bai, 2006) e submetido ao *International Journal of Computer Science and Network Security*. *Clusters* são grupos de objetos classificados com base nos princípios de maximizar a similaridade intraclasse e minimizar a similaridade interclasse. Objetos pertencentes a um mesmo *cluster* têm alta similaridade quando comparados um com outro e são bastante discrepantes em relação aos objetos de outros *clusters* (Han; Kamber, 2000).

Para descobrir os candidatos a aspectos, foram realizados procedimentos de criação de *clusters* sobre os rastros de execução do programa com o objetivo de agrupar cenários de execução com comportamento semelhante. Em seguida,

regras de associação foram detectadas entre as relações de chamadas dos métodos possibilitando que o relacionamento entre o código base e o código transversal seja identificado (He; Bai, 2006).

Na definição de regras de associação, foram considereados um conjunto de itens $I = \{i_1, i_2, \dots, i_n, \}$ e um banco de dados de transações $D$, onde cada transação $T$, associada a um identificador $TID$, é um conjunto de itens tal que $T \subseteq I$. Sejam $A$ e $B$ conjuntos de itens, uma transação $T$ é dita contida em $A$ se, e somente se, $A \subseteq T$. Uma regra de associação é uma implicação $A \Rightarrow B$, sendo $A \subset I$, $B \subset I$ e $A \cup B \neq \emptyset$. Além disso, podem ser classificadas quanto aos tipos, dimensões e níveis de abstração dos valores manipulados. Regras de associação são condições com atributos e valores que ocorrem frequentemente juntas em um determinado conjunto de dados. Formalmente, seguem o padrão $X \Rightarrow Y$, isto é, $A_1 \wedge \cdots \wedge A_m \Rightarrow B_1 \wedge \cdots \wedge B_n$, sendo $A_i$ (para $i \in \{1, \dots, m\}$) e $B_j$ (para $j \in \{1, \dots, n\}$) pares atributo-valor. A confiança de uma regra é a medida da probabilidade (ou da certeza) da proposição Y ocorrer quando a proposição X ocorrer e o suporte de um regra é a proporção de vezes que a relação ocorre (Han; Kamber, 2000).

Alguns exemplos podem ser utilizados para explicar os conceitos anteriores (Han; Kamber, 2000). Considerando o exemplo de um banco de dados de produtos eletrônicos, alguns *clusters* que podem ser identificados são: produtos telefônicos, computadores, produtos automotivos, eletrodomésticos e áudio/vídeo. Para esse o exemplo, a seguinte regra de associação pode ser encontrada:

$$idade(X, \texttt{"20-29"}) \wedge sal\acute{a}rio(X, \texttt{"2000-3000"}) \Rightarrow compra(X, \texttt{"CD player"}), [supporte = 2\%, confian\c{c}a = 60\%]$$

Isso significa que, dos compradores de produtos eletrônicos em estudo, 2% (suporte) têm idade entre 20 e 29 anos, salário entre 2 mil e 3 mil e compraram um *CD player*. Há uma probabilidade de 60% (confiança) de que um novo comprador com idade e salário desse grupo compre um *CD player*.

A granularidade desta técnica está em nível de métodos, é necessária a intervenção do usuário para inspecionar os *clusters* resultantes, o sintoma de interesse transversal procurado no código fonte é o espalhamento de código e os dados de entrada necessários para a computação desta técnica são dinâmicos.

### 3.6.2. Explicação do Funcionamento

Para obter os rastros de execução, o software é manipulado e executado para cenários e entradas específicos. Cada cenário, considerado um objeto, corresponde a uma sequência de chamadas de métodos. Os métodos executados em cada cenário são os atributos dos objetos e o número de vezes que cada método foi chamado é valor do atributo.

Seja $n$ a quantidade de cenários e $m$ a quantidade de métodos, então o conjunto de objetos é $O = (o_1, o_2, \dots, o_n)$, e cada objeto é um vetor $(a_{i1}, a_{i2}, \dots, a_{im})$ $(i = 1, 2, \dots, n)$ de dimensão $m$.

Uma aplicação bancária foi utilizada como estudo de caso para explicar o funcionamento da técnica (He; Bai, 2006). O software foi desenvolvido utilizando a tecnologia de OO e os cenários para este estudo foram (i) criação de contas, (ii) depósito, (iii) saque e (iv) balanço. Alguns cenários ($KH_1$, $KH_2$,$CK_1$, ...) e seus respectivos métodos são mostrados no Tabela 2. O número entre parênteses após o identificador do método indica a quantidade de vezes que ele foi executado. Por exemplo, no cenário $KH_1$, alguns métodos foram executados, tais como `Trans.Accept()`, `Encry.TsMac()`, `KH.hz_kh()`,

`Error.Lock()`, `Trans.Send()` e `Ans.Process()`,esses métodos foram executados apenas um vez, exceto o último que foi executado duas vezes.

Em seguida, os autores definem uma estrura chamada matriz de dados dos objetos , a qual é montada da seguinte forma: para cada objeto $o_i$ e método $a_j$, se o método $a_j$ $(j = 1, 2, \dots, m)$ é invocado pelo objeto $o_i$ $k$ vezes, então o valor da $j$-ésima posição do objeto $o_i$ é $k$, caso contrário, é zero. Também é definida a matriz de dissimilaridade entre objetos, a qual armazena a medida de dissimilaridade para cada par de objetos em uma estrutura $n * m$. Se os valores da $w$-ésima posição do objeto $o_i$ e $o_j$ são ambos diferentes de zero, significa que $o_i$ e $o_j$ invocam $a_w$. Ou seja, $o_i$ e $o_j$ têm comportamento similar em termos de $a_w$, a diferença entre eles pode ser o valor $k$. Se ambos os valores são zero, significa que nenhum deles invoca $a_w$. Caso um valor seja zero e o outro diferente de zero, significa que existe uma dissimilaridade entre eles em termos do comportamento de $a_w$.

*Clusters* na matriz de dados dos objetos e na matriz de dissimilaridade entre objetos produzem grupos de conjuntos de objetos. Elementos com atributos comuns em cada conjunto (códigos similares dos cenários) são candidatos a interesses transversais. Para o exemplo da aplicação bancária, dois *clusters* foram obtidos referentes aos módulos de balanço diário ($DLRJ_1$, $DLRJ_2$,..., $JZRJ_1$, $JZRJ_2$,...) e tarifas ($KH_1$, $KH_2$,..., $QK_1$, $QK_2$,...). Para o primeiro *cluster*, o conjunto de atributos comuns indica a existência de dois interesses transversais: i) acesso banco de dados; e ii) bloquear/desbloquear tabela. No segundo *cluster*, os interesses transversais encontrados são referentes a (i) comunicação, (ii) encriptação/decriptação e (iii) mensagens.

Tabela 2 Conjunto de Objetos (He; Bai, 2006)

| Cenários | Métodos Executados |
|---|---|
| $KH_1$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(2), KH.hz_kh(1), ..., Error.Lock(1), Trans.Send(1) |
| $KH_2$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(2), KH.zl_kh(1), ..., Trans.Send(1) |
| $CK_1$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.hz_ck(2), ..., Trans.Send(1) |
| $CK_2$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.zl_ck (2), ..., Trans.Send(1) |
| $QK_1$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.hz_qk(2), ..., Trans.Send(1) |
| $QK_2$ | Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.zl_qk(2), ..., Trans.Send(1) |
| $DLRJ_1$ | DB.Open(1), Lock.Table(5), RJ.rjemnllidr(1), Lock.UnlockTable(5), ..., DB.Close(1) |
| $DLRJ_2$ | DB.Open(1), Lock.Table(5), RJ.rjmemlcmsj(1), Lock.UnlockTable(5), ..., DB.Close(1) |
| $JZRJ_1$ | DB.Open(1), Lock.Table(8), RJ.rj1zzjzzc(1), Lock.UnlockTable(8), ..., DB.Close(1) |
| $JZRJ_2$ | DB.Open(1), Lock.Table(8), RJ.rj1hzlsz(1), Lock.UnlockTable(8), ..., DB.Close(1) |
| | .... |

Uma vez que os interesses transversais foram identificados, necessita-se apontar a sua localização. Os passos seguintes descrevem o procedimento. Seja $M = (m_1, m_2, m_3)$ uma sequência de chamada de métodos e $m_3$ indicado como interesse transversal. Cada método $m$ é considerado como uma transação da regra de associação e a sequência de métodos chamados por $m$ constitui seu conjunto de itens. Com o objetivo de identificar as posições iniciais e finais, os métodos `first()` e `end()` são adicionados ao conjunto de itens de $m$ para marcar, respectivamente, o início e fim das invocações de métodos. Um conjunto de itens $I = (i_1, i_2, \dots, i_m)$ é constituído por $m$ métodos no código fonte, cada qual considerado como um item. Como foram adicionados os métodos `first()` e `end()` ao conjunto de itens, eles recebem as respectivas notações $i_0$ e $i_{m+1}$. Portanto, a extensão do conjunto $I$ é $I^* = (i_0, i_1, i_2, \dots, i_m, i_{m+1})$.

Para a aplicação bancário, os itens são `first()`, `Trans.accept()`, `Encry.TsMac()`, ..., `DB.Close()`, `end()`, representados por $i_0, i_1, i_2, \dots, i_{19}, i_{20}$. Como cada transação é determinada por um cenário do código fonte, cada cenário recebe um $TID$ e o conjunto de transações $T = \{T_1, T_2, \dots, T_m\}$, para cada $T_j, j = 1, 2, \dots, m,\ T_j \subseteq I^*$, inclui as transações. Para a aplicação analisada, os cenários $KH_1$, $KH_2$, ..., $JZRJ_2$, ... são representados pelas transações $T_0$, $T_1$, $T_2$, ..., $T_9$, .... O conjunto de transações para a aplicação bancária é apresentado na Tabela 3. Caso algum item se repita em uma transação, ele é representado pela notação conforme o item 3 da transação $T_0$: $i_3^{'}$ e $i_3^{''}$.

Tabela 3 Conjunto de Transações (He; Bai, 2006)

| $T$ | Conjunto de Itens |
|---|---|
| $T_0$ | $i_0, i_1, i_2, i_3^{'}, i_4, i_3^{''}, i_{10}, i_{11}, i_{20}$ |
| $T_1$ | $i_0, i_1, i_2, i_3^{'}, i_5, i_3^{''}, i_{10}, i_{11}, i_{20}$ |
| $T_2$ | $i_0, i_1, i_2, i_3^{'}, i_6^{'}, i_3^{''}, i_6^{''}, i_3^{'''}, i_{10}, i_{11}, i_{20}$ |
| $T_3$ | $i_0, i_1, i_2, i_3^{'}, i_7^{'}, i_3^{''}, i_7^{''}, i_3^{'''}, i_{10}, i_{11}, i_{20}$ |
| $T_4$ | $i_0, i_1, i_2, i_3^{'}, i_8^{'}, i_3^{''}, i_8^{''}, i_3^{'''}, i_{10}, i_{11}, i_{20}$ |
| $T_5$ | $i_0, i_1, i_2, i_3^{'}, i_9^{'}, i_3^{''}, i_9^{''}, i_3^{'''}, i_{10}, i_{11}, i_{20}$ |
| $T_6$ | $i_0, i_{12}, i_{13}, i_{14}, i_{18}, i_{19}, i_{20}$ |
| $T_7$ | $i_0, i_{12}, i_{13}, i_{15}, i_{18}, i_{19}, i_{20}$ |
| $T_8$ | $i_0, i_{12}, i_{13}, i_{16}, i_{18}, i_{19}, i_{20}$ |
| $T_9$ | $i_0, i_{12}, i_{13}, i_{17}, i_{18}, i_{19}, i_{20}$ |
| | . . . |

As regras de associação são produzidas depois de fornecidos o conjunto de transações e de itens, os valores de confiança e o suporte a um algoritmo de mineração de regras de associação. Nesta análise, o suporte assegura que os candidatos a aspectos sejam frequentes, enquanto a confiança especifica a estabilidade da localização dos interesses transversais. Após a execução de um algoritmo de mineração de regras de associação, as seguintes regras foram encontradas com confiança e suporte de 40%:

(1) $se\ i_0\ então\ i_1\ \wedge\ i_2\ \wedge\ i_3$

(2) $se\ i_{10}\ \wedge\ i_{11}\ então\ i_{20}$

(3) $se\ i_0\ então\ i_{12}\ \wedge\ i_{13}$

(4) $se\ i_{18}\ \wedge\ i_{19}\ então\ i_{20}$

Pela regra (1), percebe-se que no início de alguns cenários ($KH_1$, $KH_2$, $CK_1$, $CK_2$, $QK_1$, $QK_2$), os métodos $i_1, i_2\ e\ i_3$ são executados. De forma semelhante, imediatamente antes do final de outros cenários ($DLRJ_1$, $DLRJ_2$, $JZRJ_1$, $JZRJ_2$), os métodos $i_{18}$ e $i_{19}$ são executados.

## 3.7. Comparação Entre os Métodos

Analisando as técnicas apresentadas nas seções anteriores, é possível notar que apenas as abordagens de análise dos rastros de execução e de utilização de *clusters* com regras de associação são consideradas técnicas dinâmicas. Os métodos que consideram os trechos de código clonados como indícios de interesses transversais conseguem retornar ao usuário um resultado mais detalhado, pois informam os fragmentos de código espalhados pela aplicação, os demais métodos têm granularidade em nível de métodos.

No que diz respeito aos sintomas de interesses transversais investigados pelas técnicas supracitadas, todas levam em consideração o espalhamento de código para efetuar a computação. No quesito de envolvimento do usuário, pode-se notar que em todas as técnicas é considerado escasso, há a necessidade de intrusão apenas ao final do processo.

A análise das técnicas estudadas, caracterizando-as seguindo os critérios de granularidade, tipo de dados de entrada, sintomas de interesse transversal e envolvimento do usuário, e uma breve descrição para facilitar a identificação de cada técnica são apresentadas na Tabela 4.

Tabela 4 Técnicas de Detecção de Interesses Transversais

| | **Técnica de Detecção de Interesse Transversal** | | | | | |
|---|---|---|---|---|---|---|
| **Critério** | **Análise *Fan-In*** | **Análise dos Rastros de Execução** | **Análise do Grafo de Fluxo de Controle** | **Detecção de Clones utilizando AST** | **Detecção de Clones utilizando PDG** | ***Clusters* e Regras de Associação** |
| **Descrição** | Cálcula o *Fan-In* para determinar espalhamento de código | Busca por padrões recorrentes nos rastros de execução | Busca por padrões recorrentes no grafo de fluxo de Controle | Identifica código duplicado utilizando árvore sintática abstrata | Identifica código duplicado utilizando grafo de dependência | Agrupa cenários de execução com desempenho semelhante |
| **Tipo de Análise** | Estática | Dinâmica | Estática | Estática | Estática | Dinâmica |
| **Granularidade** | Métodos | Métodos | Métodos | Fragmentos de código | Fragmentos de código | Métodos |
| **Sintoma Analisado** | Espalhamento de Código | Espalhamento de Código | Espalhamento de Código | Espalhamento de Código | Espalhamento de Código | Espalhamento de Código |
| **Envolvimento do Usuário** | Selecionar candidatos a aspectos de uma lista | Examinar padrões recorrentes | Examinar padrões recorrentes | Examinar trechos de códigos considerados clones | Examinar trechos de códigos considerados clones | Examinar *clusters* resultantes |

## 3.8. Considerações Finais

Neste capítulo, foram apresentadas algumas técnicas encontradas na literatura com o objetivo de identificar indícios de interesses transversais no código fonte de softwares orientados a objetos. Para cada abordagem estudada, os autores foram identificados, os pré-requisitos necessários para entendimento da técnica foram esclarecidos e critérios de comparação foram aplicados. Além disso, foram utilizados exemplos com o intuito de melhor ilustrar o funcionamento das técnicas.

Por causa do aperfeiçoamento e considerável desenvolvimento das pesquisas relacionadas a identificação de interesses transversais, o número de técnicas de mineração de aspectos propostas tem crescido bastante. Com o objetivo de identificar de forma mais clara e simples as semelhanças e as diferenças entre as principais técnicas, ao final do capítulo foi apresentado um quadro resumo com as categorias definidas por (Kellens *et al*., 2007) utilizadas para agrupar e comparar técnicas de mineração de aspectos.

# 4. FERRAMENTAS DE APOIO AO DESENVOLVIMENTO

## 4.1. Considerações Iniciais

A utilização de ferramentas computacionais durante as etapas de desenvolvimento de novos aplicativos é comumente incentivada e bastante difundida em razão dos benefícios que elas podem porporcionar. Um das principais vantagens é a minimização do tempo gasto no projeto, pois uma determinada função pode ser implementada utilizando um código pronto, eficiente, estável e que atende as necessidades do projeto. As ferramentas computacionais utilizadas na implementação dos *plug-ins* desenvolvidos neste trabalho são comentadas nesse capítulo.

Breve história e a arquitetura da plataforma IDE[2] Eclipse são resumidas na Seção 4.2. O *plug-in* Java *Development Tools* é apresentado na Seção 4.3. Ferramentas computacionais utilizadas para a implementação dos *plug-ins* são descritas na Seção 4.4.

## 4.2. Plataforma IDE Eclipse

A empresa *Object Technology International* (OTI) foi a responsável pelo desenvolvimento do código Java que forma a base do Eclipse. Posteriormente, em meados de 1996, a *International Business Machines* (IBM) comprou a OTI para dar início aos trabalhos sobre o código do Eclipse visando a integração de suas diversas ferramentas de acordo com os padrões definidos pelo *Object Management Group* (OMG), o qual produz e mantém especificações de interoperabilidade entre aplicações (Geer, 2005).

Para a comunidade de desenvolvimento de software, o termo Eclipse normalmente está associado com o *Eclipse Software Development Kit* (Eclipse

[2] *Integrated Development Environment*

SDK), líder entre os ambientes integrados de desenvolvimento Java (IBM, 2006), e com a ferramenta para construção de produtos baseados na plataforma Eclipse. O Eclipse SDK (doravante, Eclipse) é uma combinação de esforços de diversos projetos, incluindo *Java Development Tools* (JDT) e *Plug-in Development Environment* (PDE). O Eclipse *Platform* (doravante, Plataforma) é uma composição de componentes com as funções necessárias para construir uma IDE.

Uma das principais vantagens em utilizar a Plataforma é a possibilidade de integração entre diversas ferramentas e de executar em múltiplos sistemas operacionais. Por exemplo, a Plataforma torna-se uma IDE para desenvolvimento Java adicionando o *plug-in* JDT, assim como se torna um ambiente para desenvolvimento C/C++ ao adicionar o *plug-in* C/C++ *Development Tools* (CDT). Esta característica pode ser muito útil principalmente para organizações que desejam integrar grandes projetos que utilizam várias tecnologias. *Plug-ins* são pacotes estruturados de código-fonte e/ou dados que contribuem para a funcionalidade do software, representando códigos de bibliotecas (classes Java com API[3] pública), extensões da plataforma e, até mesmo, documentação. Além disso, *plug-ins* podem definir pontos de extensão, isto é, locais bem definidos em que outros *plug-ins* podem adicionar funcionalidade (Kawakami, 2007).

A Plataforma é construída com um mecanismo de descoberta, integração e execução de *plug-ins* que seguem especificações *Open Services Gateway initiative* (OSGi[4]). Um desenvolvedor cria ferramentas adicionais que operam em arquivos de configuração do *workspace* e *workbench* e as insere no diretório de *plug-ins* do Eclipse. Quando a Platforma é executada, o diretório é verificado

[3] *Application Programming Interface*
[4] http://osgi.org

pelo *Platform Runtime* e os arquivos de configuração dos *plug-ins* são lidos para que sejam construídos seus registros em memória. Ao final do processo de iniciação, o usuário é provido de um ambiente de desenvolvimento composto de um conjunto de *plug-ins*.

Para desenvolver interfaces gráficas personalizadas, o Eclipse é equipado com os *toolkits Standar Widget Toolkit* (SWT) e *JFace* fornecidos pelo *workbench*. A estrutura do Eclipse com esses componentes é ilustrada na Figura 8. Essa figura inclui a plataforma básica do Eclipse e ferramentas úteis para o desenvolvimento de *plug-ins* (JDT), que implementa um ambiente de desenvolvimento Java completo, e o ambiente desenvolvedor de *plug-in* (PDE), que acrescenta ferramentas especializadas para organizar o desenvolvimento de *plug-ins* e extensões. Os componentes nomeados "Plug-in" representam outros *plug-ins* desenvolvidos e conectados ao Eclipse com o objetivo de estender sua funcionalidade e atender solicitações de um projeto ou ambiente de trabalho específicos.

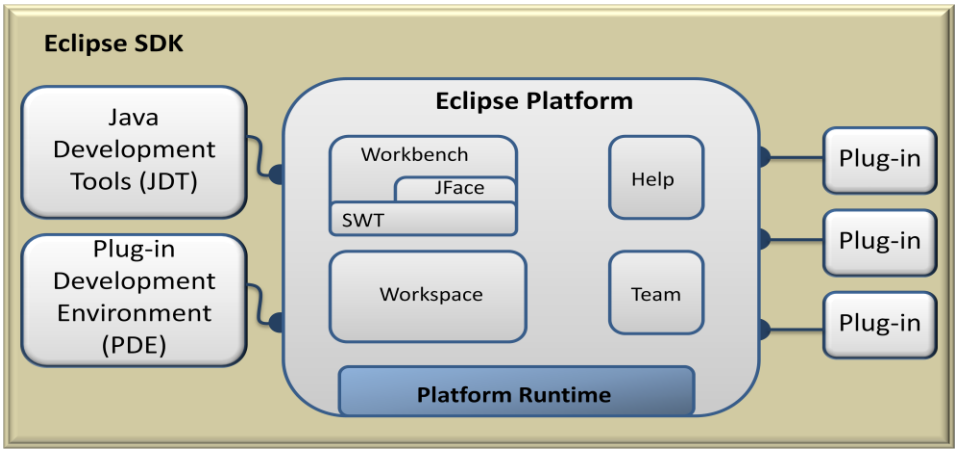


Figura 8  Arquitetura da Plataforma Eclipse

### *4.3. Plugin Java Development Tools*

O *plug-in* JDT está incluído no Eclipse e adiciona capacidades à Plataforma transformando-a em ambiente de desenvolvimento Java com diversos recursos. A seguinte lista sumariza algumas funções implementadas pelo JDT (JDT, 2011):

- Organizar códigos fonte Java, bibliotecas de arquivos .jar (*Java ARquive* - JAR) e outros recursos necessários aos usuários em estruturas de projetos. Além disso, isolar os arquivos binários (.class) em de diretórios exclusivos;
- Permitir navegação em projetos Java em termos de pacotes, classes, métodos e atributos;
- Dispor de um editor de código fonte com diferentes marcações para palavras reservadas da linguagem de programação Java e identificadores de classes, de atributos e de métodos criados pelo desenvolvedor. Fornecer, ainda, eficiente visualização de erros e avisos de compilação utilizando anotações na margem do editor;
- Possuir API de ajuda para exibir documentação no padrão Javadoc de elementos selecionados pelo usuário, ferramentas de *auto-complete* e assistentes para criar e organizar declarações de *import* de bibliotecas;
- Fornecer ferramentas de refatoração de código para extração e edição de assinatura de métodos, atualização de referências e pré-visualização de modificações;
- Acionar o compilador Java automaticamente após modificações no código fonte e informar erros em um painel específico;
- Executar programas Java possibilitando comunicação do usuário com fluxos de entrada e saída mediante o uso do console;
- Fornecer ferramenta de *debug* que permita adicionar pontos de interrupção em métodos e inspeção de atributos e variáveis locais.

O JDT é implementado no Eclipse por um conjunto de *plug-ins* em uma arquitetura que garanta a separação entre as unidades de interface com usuário e as unidades do núcleo do JDT (IBM, 2006). Esta arquitetura e as principais conexões entre a Plataforma e o JDT é ilustrada na Figura 9.

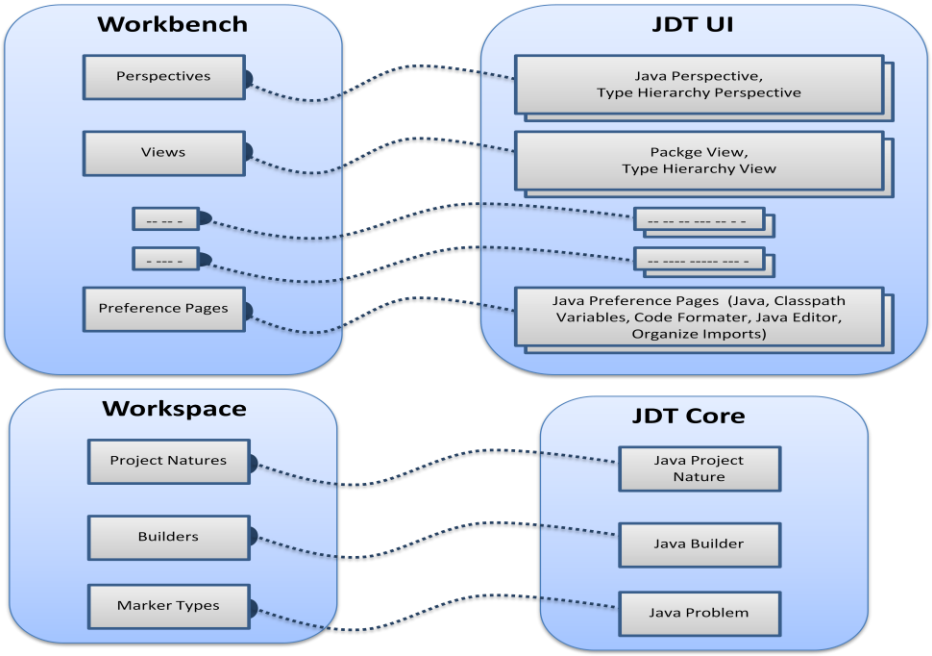


Figura 9 Conexões entre o Plataforma e o JDT (Chapman, 2006)

A estrutura *JDT UI* (`org.eclipse.jdt.ui`) presente nessa arquitetura é a unidade responsável por tratar a comunicação com o usuário. Ela define áreas ou visões onde o usuário pode editar o código fonte Java, visualizar informações dos projetos, pacotes e bibliotecas JAR, criar novos projetos, alterar atributos de configuração, dentre outras funções. Ao *JDT Core* (`org.eclipse.jdt.core`), compete executar funções mais árduas como compilação, *debug* e refatoração. Além disso, sua utilização não é dependente de uma biblioteca gráfica em particular, o que possibilita seu aproveitamento como um módulo separado do JDT.

## 4.4. Outras Ferramentas

Durante o desenvolvimento dos *plug-ins*, alguns softwares com funções específicas foram utilizados para implementar pontos particulares do projeto, com o objetivo de alcançar melhores resultado e abreviar o tempo de desenvolvimento. Foram desenvolvidos três *plug-ins* que correspondem a três técnicas semânticas de mineração de indícios de interesses transversais em softwares orientados a objetos.

Para implementação da técnica detecção de clones utilizando árvore de sintática abstrata, foi utilizado o software JCCD[5] (*A Flexible Java Code Clone Detector API*) (Biegel; Diehl, 2010) para encontrar os *Trechos Clones* no código fonte (trechos de código similares). Em seguida, as métricas descritas no Capítulo 3 foram aplicadas para encontrar os candidatos a aspectos. O JCCD é baseado em uma arquitetura de *pipeline* criado a partir da AST, tal arquitetura permite substituir qualquer parte do processo de detecção de clones sem alterar o *pipeline* como um todo. Esta característica possibilita utilizá-lo na implementação de novas técnicas de detecção de clones, bem como avaliar novos conceitos relacionados ao assunto. Além disso, o JCCD utiliza pré-processadores e comparadores em diferentes fases do *pipeline*. Pré-processadores são utilizados para generalizar o código fonte original por meio da deleção de nós da árvore e comparadores são utilizados para identificar declarações de métodos semelhantes ignorando seus nomes.

Outra ferramenta utilizada foi Zest[6] (Bull *et al*., 2004) que consiste em um conjunto de componentes de visualização construídos especialmente para o Eclipse. Zest foi desenvolvida utilizando componentes gráficos SWT/Draw2D (bibliotecas gráficas) presentes no Eclipse, o que permite uma integração

[5] http://sourceforge.net/projects/jccd/files/jccd-1.0.6.tar.gz/download
[6] http://www.eclipse.org/gef/zest/

perfeita com o ambiente de desenvolvimento. O principal objetivo da Zest é proporcionar rápida e simples criação de ferramentas para visualização de grafos os quais podem ser apresentados em diferentes *layouts*, abrangendo estrutura hirárquia de árvore, estrutura radial, em grade e *spring*. Zest foi utilizado para visualizar a estrutura de chamadas de métodos das técnicas *Fan-In* e Análise do Grafo de Fluxo de Controle. Os resultados da técnica *Fan-In* são exibidos de forma que cada nó do grafo representa um método chamado e os métodos declarados dentro da mesma classe são dispostos com a mesma cor. De forma semelhante à implementação da técnica *Fan-In*, na Análise de Grafo de Fluxo de Controle, os métodos do software são representados em forma de nós do grafo, sendo que cada cor representa a classe onde o método foi declarado. Um exemplo de como a Zest foi utilizada é apresentada na Figura 10.

Para uma melhor visualização dos resultados da técnica de Análise do Fluxo de Controle foi utilizada a ferramenta *Visualiser*, presente no *plugin* AJDT (*AspectJ Development Tools*) desenvolvido para o Eclipse. O AJDT está para o desenvolvimento de projetos usando a linguagem de programação AspectJ assim como o JDT está para projetos Java. As funções mais importantes adicionadas por esse *plug-in* estão relacionadas à visualização do comportamento tranversal das linguagens de programação orientadas a aspectos, incluindo ambientes para visualizar informações pontuais do software e informações em um contexto global, possibilitando uma visão geral do software.

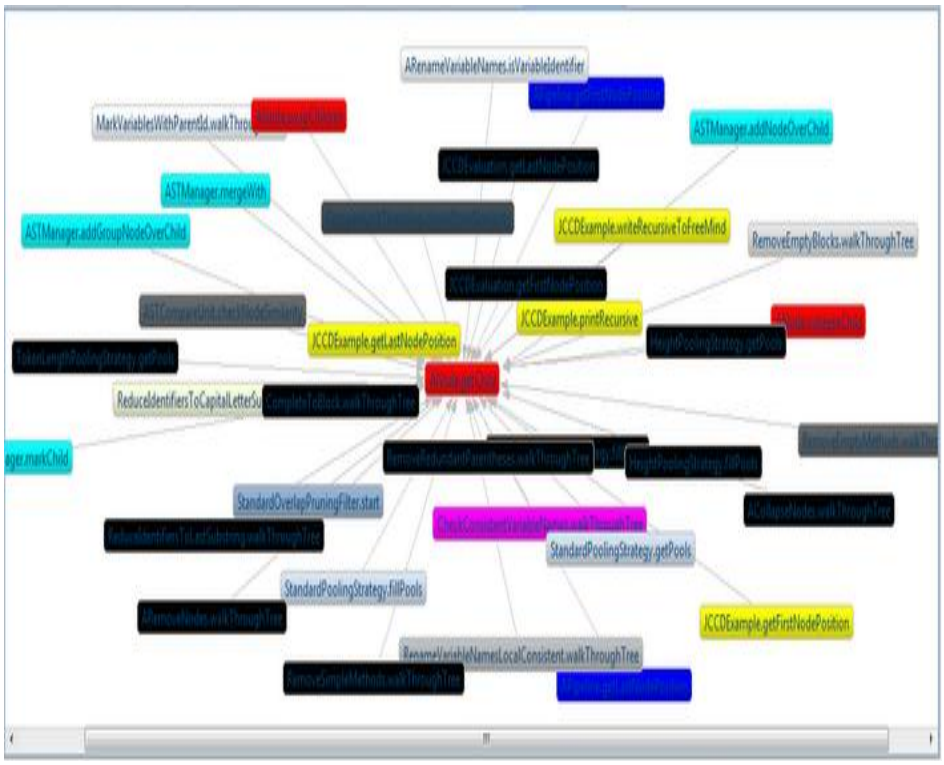


Figura 10 Visualização de Métodos com a Ferramenta Zest

A *Visualiser* é uma *View* do Eclipse que provê uma apresentação visual de um projeto selecionado consistindo de uma área com várias colunas dispostas verticalmente, sendo cada coluna relacionada a um arquivo fonte. O tamanho das colunas é proporcional à quantidade de linhas de código do arquivo e listras são inseridas em posições apropriadas para representar os locais no código onde um *advice* está atuando. No entanto, para o desenvolvimento dos *plug-ins* propostos neste trabalho, a *Visualiser* foi utilizada com o intuito de evidenciar *Trechos Clones* entre si, o que permite ao usuário ter uma visão de como os clones afetam o software. Um exemplo da utilização da Visualiser é apresentada na Figura 11.

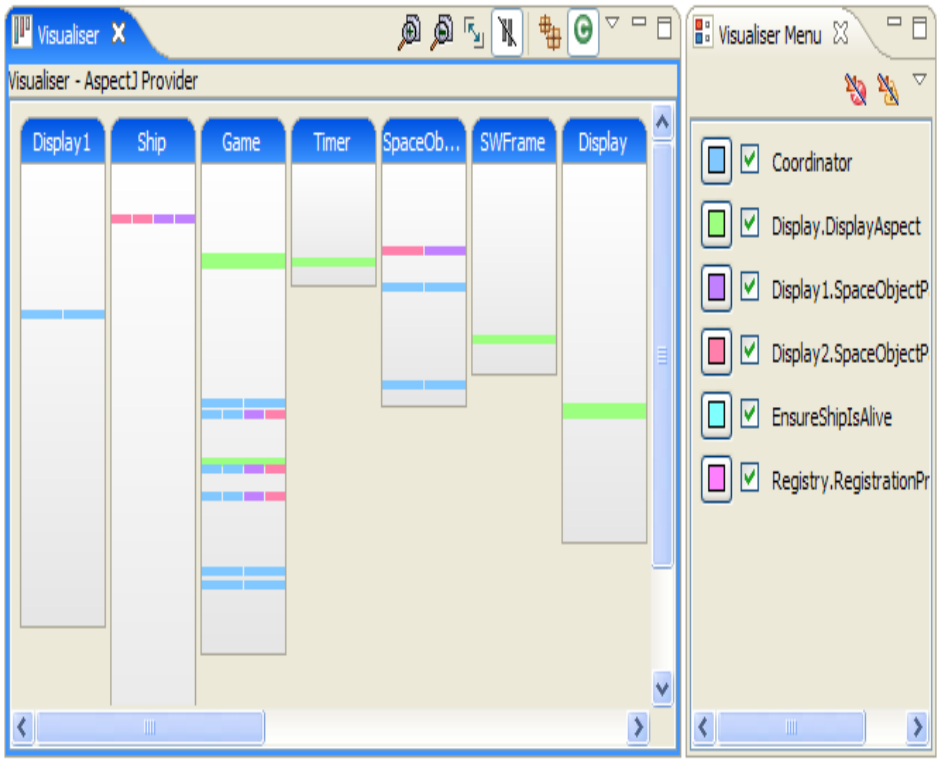


Figura 11 Visualizador de Aspectos (Chapman, 2006)

## 4.5. Considerações Finais

Este capítulo apresentou os principais componentes do ambiente de desenvolvimento Eclipse dando ênfase aos conceitos de Eclipse SDK e Eclipse *Platform*, bem como a forma de integração e de execução de *plug-ins* utilizada pelo Eclipse *Platform*. Além disso, foram citados os principais componentes de criação de interfaces gráficas disponíveis no Eclipse: SWT e JFace.

Foram apresentados neste capítulo breve resumo sobre o início do processo de desenvolvimento do código fonte do eclipse e as empresas envolvidas. Posteriormente, o *plug-in* para desenvolvimento Java (JDT) foi apresentado e suas principais características evidenciadas, ressaltando sua arquitetura e sua forma de integração com a Plataforma

Por fim, foram descritas as ferramentas com funções específicas utilizadas durante as etapas de desenvolvimento deste trabalho. Uma ferramenta para detecção de clones em código fonte Java (JCCD), uma biblioteca de componentes gráficos para o Eclipse (Zest), e uma ferramenta para visualizar código fonte em forma de barras e listras (View Visualiser).

# 5. TRABALHOS RELACIONADOS

Percebe-se que há algumas referências na literatura sobre ferramentas para mineração de aspectos. Este capítulo apresenta, sucintamente, alguns trabalhos relacionados.

William G. Griswold, Jimmy J. Yuan e Yoshikiyo Kato criaram uma das primeiras ferramentas com o propósito de identificação de interesses transversais no código fonte, denominada *Aspect Browser* (Griswold *et al*., 2001). *Aspect Browser* utiliza um padrão léxico para consultar o código fonte e mapas para manipulação e representação de resultados. Além disso, *Aspect Browser* usa o *Grep* internamente viabilizando consultas complexas por meio de expressões regulares, o que o torna uma ferramenta poderosa, mesmo utilizando padrões léxicos do código fonte. Para utilizar a *Aspect Browser*, o usuário deve fornecer uma sequência de caracteres a qual deseja consultar e, em seguida, ela apresenta ao usuário uma visão do software por meio de barras verticais indicando os arquivos que contêm a sequência fornecida como entrada. Uma visão geral da ferramenta é apresentada na Figura 12.

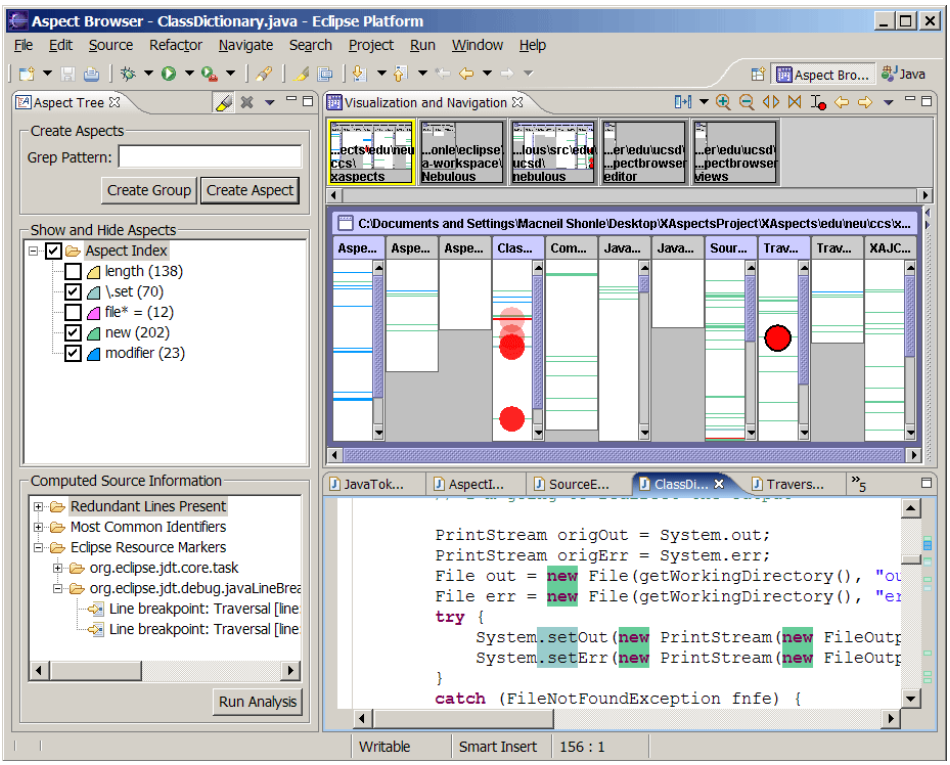


Figura 12 Visão Geral do Aspect Browser (Griswold et al., 2005)

É possível notar na parte esqueda uma *View* chamada *Aspect Tree* onde o usuário pode criar/editar aspectos encontrados a partir da expressão *Grep* informada; nessa *View*, informações sobre o código fonte em análise são apresentadas. Na parte central, uma representação gráfica em alto nível dos pacotes e arquivos do software é apresentada na *View Visualization and Navigation*, o que permite ao usuário analisar a modularização e o entrelaçamento de código fonte. Além disso, o editor de código fonte é apresentado, destacando o trecho que o usuário analisa.

O software chamado AMT (*Aspect Mining Tool*) desenvolvido por Jan Hannemann e Gregor Kiczales analisa o código fonte buscando por padrões léxicos (Hannemann; Kiczales, 2001). No entanto, o AMT permite realizar

consultas estruturais considerando tipos de dados encontrados no código. Tal abordagem é um avanço com relação ao *Aspect Browser*, mas não é estável, pois, se muitas instâncias de um mesmo tipo forem utilizadas para diferentes propósitos, essa abordagem torna-se insatisfatória. Além das análises léxicas e estruturais, o ATM pode ser extendido com outros tipos de análises, por exemplo, análises baseadas na assinatura. Assim, o AMT é considerado um conjunto de ferramentas multi-modal, o que permite o uso combinado de várias técnicas. Cada análise funciona como uma consulta e o resultado é visualizado no código fonte como um conjunto de linhas semelhantes.

A ferramenta FEAT (*Feature Exploration and Analysis Tools*) desenvolvida por Martin P. Robillard e Gail C. Murphy permite a localização e classificação de interesses transversais no código fonte por meio de uma abordagem denominada *Concern Graphs* (Robillard; Murphy, 2007). Para tanto, ela utiliza tipos de dados, métodos e atributos encontrados no código fonte na geração do seu modelo. A FEAT foi implementada como um *plug-in* para o Eclipse e uma de suas principais vantagens é a possibilidade de ser utilizada para registrar conhecimento sobre o código fonte. Uma vez criados, os interesses transversais podem ser armazenados e associados entre si verificando suas interações e existência de interdependência. A FEAT no Eclipse é apresentada na Figura 13.

A *View Concern Graph* apresentada na parte superior esqueda da Figura **13** mostra hierarquicamente os interesses transversais e permite ao usuário editá-los. Ao selecionar um interesse transversal, os seus participantes são apresentados na *View Participants*, na parte superior central da Figura **13**. Após selecionar um item na *View Participants*, o usuário pode utilizar a *View Projection* para adicionar relacionamentos entre o participante atual e qualquer outro participante do interesse transversal selecionado na *View Concern Graph*.

Além disso, com o código fonte do participante selecionado é apresentado em um editor de códigos na parte inferior da Figura 13.

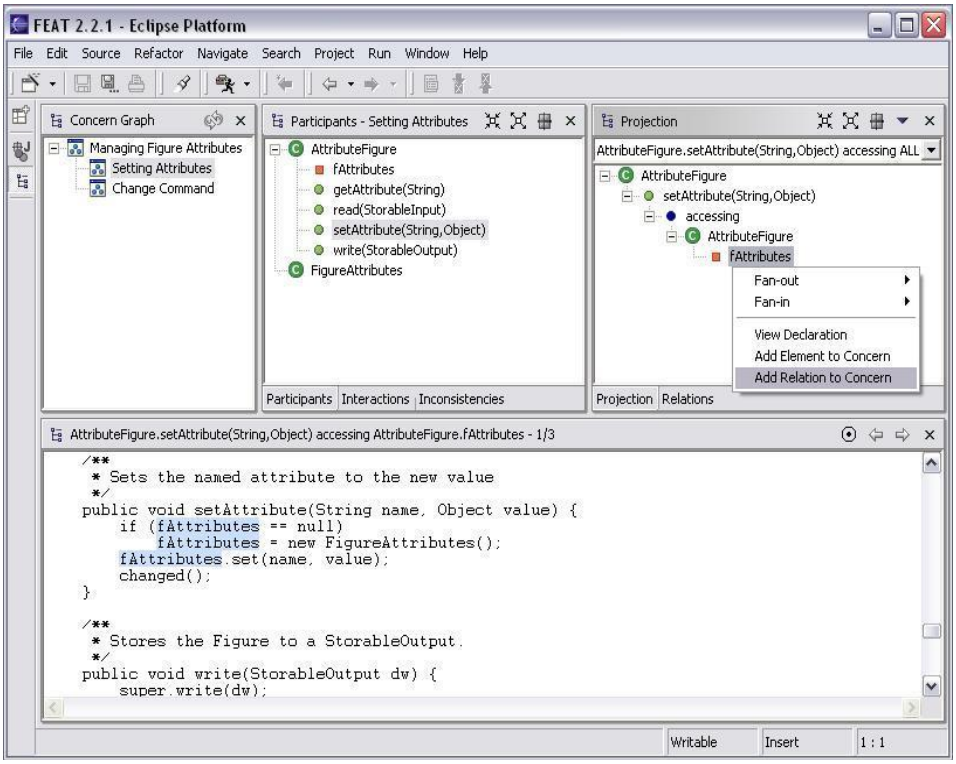


Figura 13 Visão Geral da Ferramenta FEAT (Novais, 2009)

Outra ferramenta criada para o Eclipse é chamada FINT, seus autores são Marius Marin, Leon Moonen e Arie van Deursen (Marin *et al*., 2006), com a qual a ferramenta desenvolvida neste trabalho compartilha várias semelhanças. A FINT é mantida pelos integrantes do SERG (*Software Engineering Research Group*) sob a liderança de Arie van Deursen e a versão mais recente é o FINT 0.6. A FINT implementa três técnicas de análise de código: i) *Fan-In*; ii) *Grouped Call*; e iii) *Redirections Finder*. As duas primeiras relacionadas com a identificação de interesses transversais implementados por chamadas de métodos espalhadas pelo código fonte do software, como *logging*, persistência e tratamento de exceções. A *Redirections Finder* está relacionada com a busca por

classes cujos métodos redirecionam chamadas para métodos da mesma ou de outras classes. A FINT como parte do Eclipse é apresentada na Figura 14.

A área de visualização do Eclipse é dividida pelo FINT de modo a apresentar as *Views* das três técnicas implementadas no *plug-in* e uma *View* para o editor de código Java, no qual o usuário pode vizualizar a declaração de um determinado método ao selecionar uma das *Views*. Para a técnica *Redirections Finders*, são apresentadas as classes redirecionadoras e receptora do redirecionamento como raízes de conjunto de métodos dessas classes, em um relacionamento um-para-um. Os resultados da técnica *Grouped Call* são apresentados utilizando a *view Cloned Calls*, onde os métodos são agrupados de forma que os nós em uma mesma hierarquia são os métodos chamados pelo métodos na hierarquia imediatamente superior. Na *view* da técnica *Fan-In*, os métodos são agrupados de forma que os nós filhos são métodos chamados pelo método na hierarquia superior.

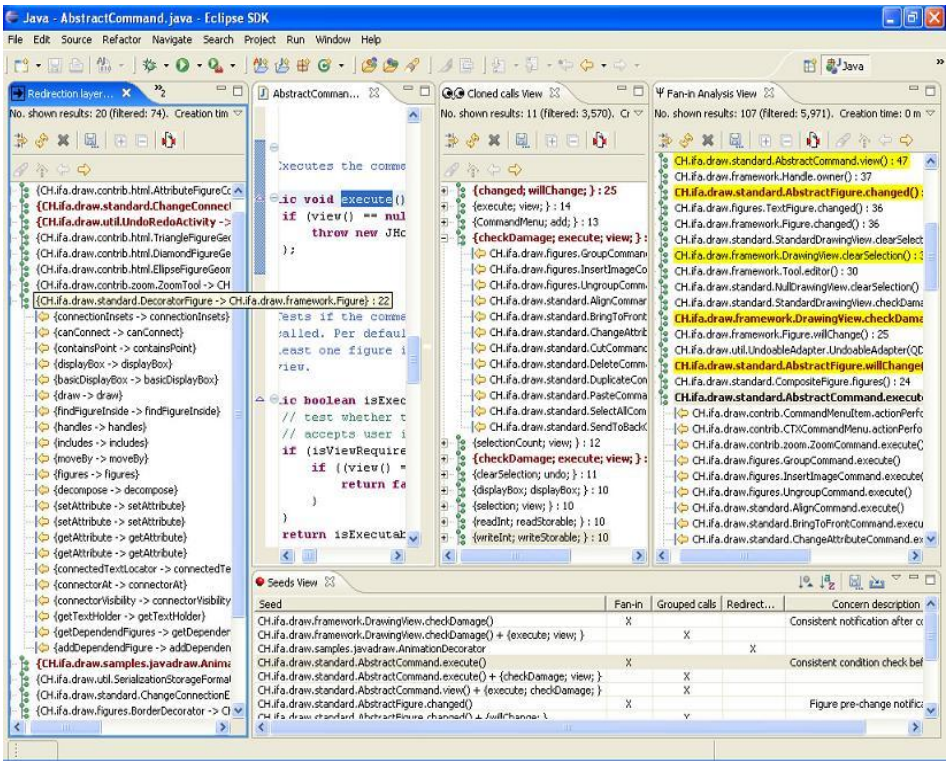


Figura 14 Visão Geral do FINT (Marin, 2008)

Analisando os trabalhos apresentados, é possível notar que seus desenvedores buscam sempre utilizar ferramentas existentes e bem fundamentadas como base para seus trabalhos. Isso pode ser percebido pelo fato de várias ferramentas utilizarem o Eclipse como plataforma base, uma vez que este fornece diversas facilidade relacionadas à inspeção do código fonte em análise.

# 6. APOIOS COMPUTACIONAIS PARA IDENTIFICAR INDÍCIOS DE INTERESSES TRANSVERSAIS EM CODIGO FONTE JAVA

## 6.1. Considerações Iniciais

A identificação de indícios de interesses transversais em softwares implementados na linguagem de programação Java requer habilidade e conhecimento por parte do engenheiro de software envolvido (Kawakami, 2007). Assim, no Capítulo 3, algumas das principais técnicas semânticas para detecção de indícios de interesses transversais foram descritas e estudadas de forma a fornecer maior entendimento de suas características e explicar como podem ser utilizadas para obter conhecimento sobre o software em análise.

De forma a agilizar e facilitar o processo de entendimento do software, foi desenvolvido neste trabalho um conjunto de ferramentas computacionais para auxiliar engenheiros de software e desenvolvedores a descobrirem funções do software que podem estar espalhadas pelo código fonte de seus módulos. Três técnicas semânticas de mineração de aspectos foram escolhidas e implementadas em forma de *plug-ins* para o Eclipse: i) Análise *Fan-In*; ii) Análise do Grafo de Fluxo de Controle; e iii) Detecção de Clones utilizando AST.

A decisão de quais técnicas seriam implementadas foi tomada considerando o tipo de dado necessário para a sua computação. Considerando que o código fonte do software é um dos elementos a ser estudado para encontrar os interesses transversais presentes no código fonte, as três técnicas de análise estática foram selecionadas para implementação. Além disso, a existência de ferramentas para auxiliar no processo de desenvolvimento influênciou a decisão.

Uma visão geral dos apoios computacionais (*plug-ins*) desenvolvidos é apresentada na Seção 5.2. Algumas características de implementação comuns aos *plug-ins* implementados são detalhadas na Seção 5.3. A funcionalidade disponível nos *plug-ins* denominados *Fan-In*, *Flow Graph* e *AST Clone* são apresentadas na Seção 5.4, na Seção 5.5 e na Seção 5.4, respectivamente.

## 6.2. Visão Geral dos Apoios Computacionais

Os apoios computacionais desenvolvidos foram implementados na linguagem de programação Java e integrados ao Eclipse como *plug-ins*. As principais razões que justificam essa integração são:

- **Utilização de bibliotecas**. O Eclipse contém grande número de bilbiotecas que podem ser utilizadas por outros softwares caso sejam implementados como *plug-ins* deste ambiente. Além disso, utilizar o Eclipse e suas bibliotecas para implementar novos produtos possibilita adaptação rápida do usuário à nova ferramenta, visto que ele está ambientado com a aparência e característica do Eclipse;
- **Uso de extensões**. É possível desenvolver novas funções e ao mesmo tempo conseguir economia na quantidade de linhas de código utilizando pontos de extensão definidos por outros *plug-ins* do Eclipse;
- **Análise sintática do código-fonte**. Utilizando classes do Eclipse e do JDT, é possível percorrer uma estrutura de dados similar a da AST sem que seja necessária a criação de um compilador para a sua construção e a sua manipulação. Com isso, a implementação é agilizada e a confiabilidade do apoio computacional é maior, pois as classes do Eclipse e de seus *plug-ins* estão estáveis;
- **Ambiente de desenvolvimento amplamente utilizado**. O Eclipse é um dos ambientes de desenvolvimento mais utilizados pela comunidade de

desenvolvedores em Java, o que favorece a utilização do apoio computacional.

A instalação dos apoios computacioanis é simples, entretanto depende da instalação prévia dos componentes *Java™ 2 Plataform Standard Edition 6.0 Development Kit* (JDK 6.0), IDE Eclipse 3.4 e Plug-in AJDT 2.1.

Os três apoios computacionais desenvolvidos foram implementados na forma de *plug-ins* que trabalham independentemente:

- ***br.ufla.dcc.plugin.ASTClone***. Esse *plug-in* implementa a técnica de detecção de indícios de interesses transversais por meio da detecção de clones utilizando árvore sintática abstrata, doravante *Plug-in AST Clone*;
- ***br.ufla.dcc.plugin.FanIn***. Esse *plug-in* implementa a técnica de detecção de indícios de interesses transversais utilizando a métrica *Fan-In*, doravante *Plug-in Fan-In*;
- ***br.ufla.dcc.plugin.FlowGraph***. Esse *plug-in* implementa a técnica de detecção de indícios de interesses transversais utilizando grafo de fluxo de controle, doravante *Plug-in FlowGraph*.

## 6.3. Características de Implementação Semelhantes aos *Plug-ins*

Para possibilitar que o usuário acione as suas funções, os três *plug-ins* as disponibilizam por meio da implementação da interface `IworkbenchWindowActionDelegate`, a qual pertence ao pacote `org.eclipse.ui` e é disponibilizada pela extensão *org.eclipse.ui.actionSets*. As ações são apresentadas ao usuários em forma de botões presentes na barra de ferramentas principal do eclipse.

Para utilizar os *plug-ins* e iniciar a detecção dos indícios de interesses transversais de forma correta, é necessário ter um projeto selecionado no *View*

*Package Explorer*. Após selecionar um projeto, pode-se selecionar a técnica de detecção desejada por pressionar o botão de ação correspondente. Em seguida, o projeto selecionado pelo usuário é capturado e inicia-se o processo de detecção. Cada *plug-in* implementado possui *Views* utilizados para apresentar os resultados ao usuário. Antes de iniciar a análise do código fonte, o *plug-in* instancia o *View* correspondente e mostra-o no Eclipse. Dessa forma, ao concluir o processamento do código, os resultados podem ser exibidos no *View* visível ao usuário.

Com o objetivo de promover a separação entre o código fonte de implementação das ações e o código fonte correspondente à implementação das técnicas, os *plug-ins* utilizam objetos controladores que recebem a instância do projeto selecionado pelo usuário e passa-a ao objeto responsável por analisar o código fonte. Desta forma, o código fonte da técnica é completamente independente da forma como o usuário interage .

### *6.4. Plug-in Fan-In*

O *Plugin Fan-In* analisa o projeto escolhido pelo usuário de forma semelhante a um visita em pré-ordem nos arquivos do projeto. Em cada arquivo, o *plug-in* recupera as classes Java e seus respectivos métodos. Em seguida, cada corpo de método é analisado de forma a verificar e armazenar as chamadas de métodos existentes. Feito isso, o *plug-in* analisa a hierarquia de classes para calcular o *Fan-In* dos métodos chamados polimorficamente e, em seguida, marcadores são criados no editor de código fonte Java na linha correspondente ao local de declaração do método. Desta forma, ao navegar pelo código fonte, o usuário pode perceber os métodos que possuem alto valor de *Fan-In* e seu respectivo número de chamadores. Um exemplo de marcador no editor de código fonte para destacar um método com alto valor de *Fan-In* é apresentado na Figura 15. O destaque é dado ao método `getStudents()` da classe

`Teacher` declarado na linha 22. Ao posicionar o ponteiro do *mouse* sobre o marcador, uma mensagem é apresentada mostrando a quantidade de chamadores do método.

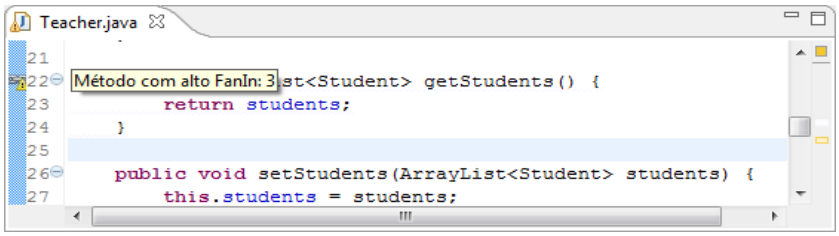


Figura 15 Marcador do Plug-in Fan-In

Além das informações no editor de código Java, o *Plug-in Fan-In* fornece outras formas para visualizar os resultados. Dois *Views* são criados no Eclipse os quais apresentam: i) informações gerais sobre os resultados; e ii) informações para um método e seu conjunto de chamadores. O primeiro *View* tem o título *Methods View* e suas principais informações são apresentadas em duas listas. A primeira lista mostra o conjunto de métodos com maiores valores de *Fan-In*, limitada a 10% do total de métodos encontrados no software. Ao selecionar um método desta lista, a segunda lista é preenchida com os chamadores do método selecionado. Além dessas informações, o *Methods View* apresenta o total de métodos com alto valor de *Fan-In*, o número total de chamadores para cada método e o número de classes distintas entre os chamadores. O *Methods View* após a análise de um determinado projeto é apresentado na Figura 16.

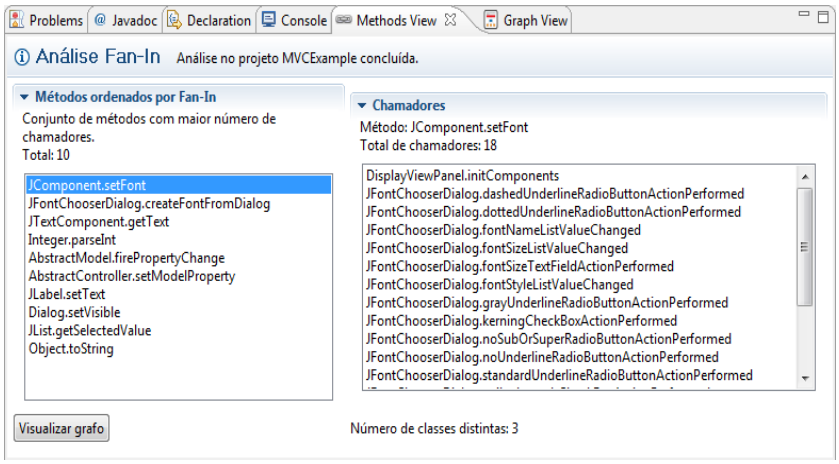


Figura 16 Methods View no Eclipse

Há também no primeiro *View* uma opção para visualizar o grafo que representa um método e seus chamadores. Ao selecionar um método da primeira lista, o segundo *View* é apresentado ao usuário (*Graph View*) o qual exibe um grafo, cujos nós são caixas de texto com o nome do método e da classe onde foi declarado, seguindo o padrão `NomeDaClasse.nomeDoMétodo`. O nó central é o método escolhido pelo usuário e os nós que o cercam são seus respectivos chamadores. Os nós coloridos de uma mesma cor represetam métodos declarados em uma mesma classe. Um caso em que o *Graph View* é utilizado para exibir o grafo de chamadores de um determinado método é apresentado na Figura 17.

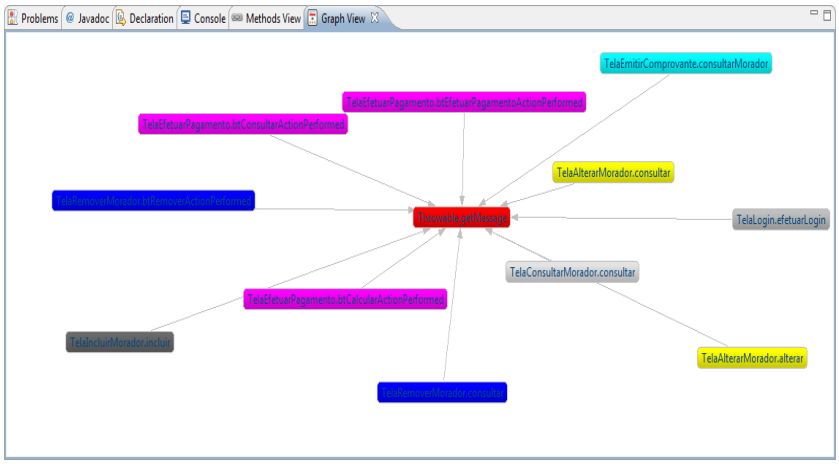


Figura 17 Graph View no Eclipse

Além de colocar o *Graph View* em primeiro plano, para o usuário visualize o grafo, o botão “Visualizar Grafo” faz com que o método escolhido pelo usuário seja apresentado no editor de código Java. Caso o método não tenha sido declarado pelo desenvolvedor, uma mensagem é apresentada informando que o método não está declarado no projeto. Tal situação ocorre, por exemplo, quando um método da API Java possui um alto *Fan-In* e é detectado pela análise. Como o código fonte de um método da API Java pode não ser encontrado no projeto do usuário, ele não pode ser visualizado no editor. Dessa forma, uma mensagem é mostrada ao usuario informado que o método não foi declarado por ele.

### *6.5. Plug-in Flow Graph*

A análise inicial de código no *Plug-in Flow Graph* é realizada por meio de uma busca em pré-ordem aos arquivos do projeto, verificando os métodos das classes. No corpo de métodos, as chamadas internas e externas são verificadas e armazenadas. Após o cálculo do número de relações internas e externas, este *plug-in* cria marcadores no editor de código fonte para facilitar a visualização do usuário quando estiver navegando pelo código. O valor *Flow Graph* informado

no marcador é calculado considerando a média entre relações internas e externas. Este é o mesmo cálculo feito para ordenar os métodos para posterior apresentação na lista de métodos com maior característica transversal. Um marcador no editor de código é apresentado na Figura 18.

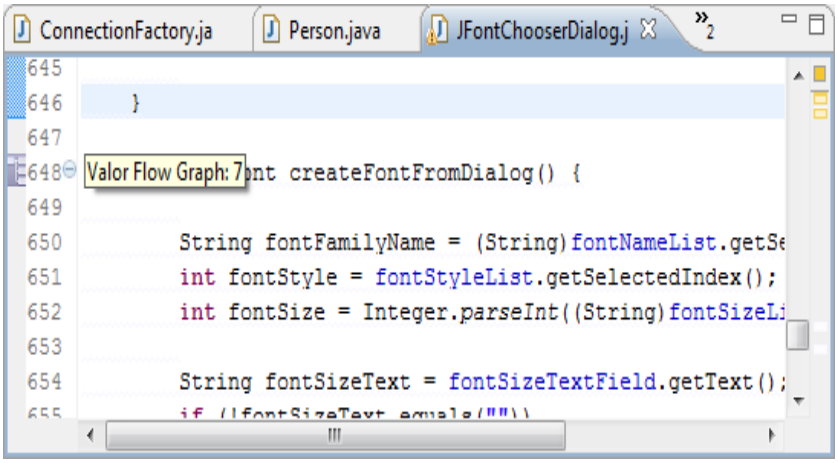


Figura 18 Marcadores do Plug-in Flow Graph

Os dois *Views* criados pelo *Plug-in Flow Graph* são semelhantes aos criados pelo *Plug-in Fan-In*. O primeiro *View* apresenta uma visão geral dos resultados (*Methods Count View*) e o segundo exibe informações para um determinado método selecionado pelo usuário (*Flow Graph View*). O *Methods Count View* segue a forma apresentada na Figura 19, uma lista contendo os métodos com comportamento transversal é apresentada, campos de texto informam a quantidade de relações internas e externas para cada método selecionado pelo usuário e o número de classes distintas dentro de cada relação.

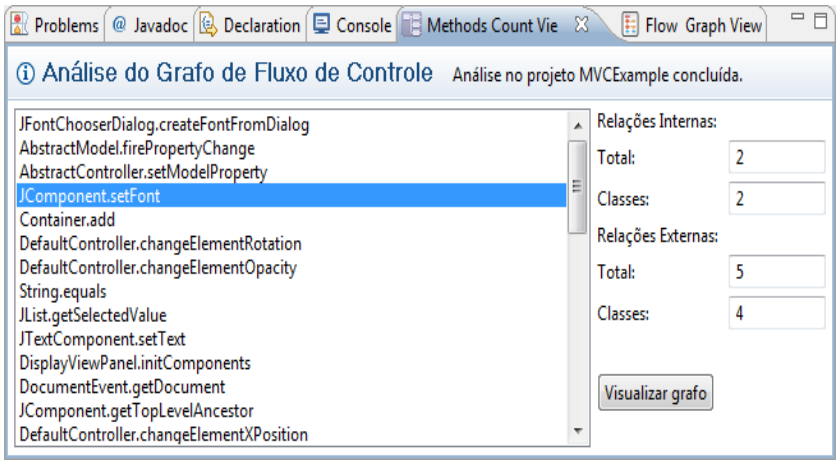


Figura 19 Methods Count View no Eclipse

Ao escolher um método da lista e selecionar o botão “Visualizar Grafo”, o *Flow Graph View* é colocado em primeiro plano para o usuário analise o grafo, cujos métodos, que compõem uma relação interna com o método selecionado, possuem arestas apontando para ele e métodos que compõem relações externas são ligados ao método selecionado por arestas saindo dele (Figura 20). Além de mostrar ao usuário o grafo das relações internas e externas, o botão “Visualizar Grafo” é responsável por fazer com que o método selecionado pelo usuário seja apresentado no editor de código fonte, como é feito no *Plug-in Fan-In*.

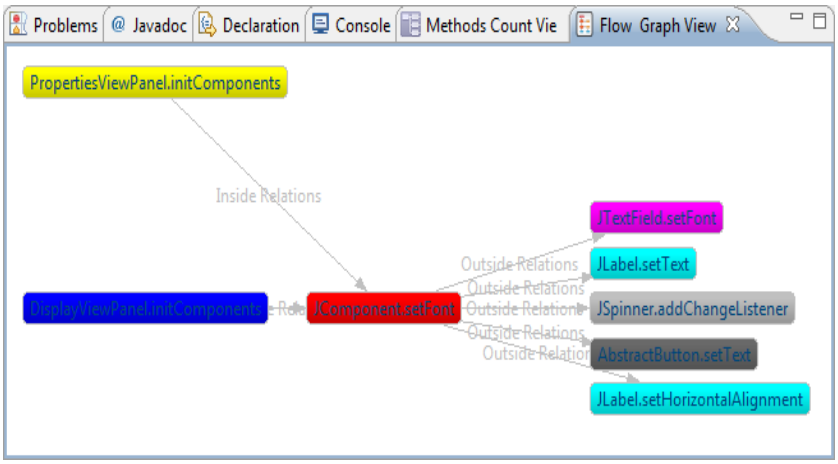


Figura 20 Marcadores do Plug-in Flow Graph

### *6.6. Plug-in AST Clone*

O *Plug-in AST Clone* inicia sua execução de forma semelhante à execução dos *plug-ins* descritos anteriormente. No entanto, durante a análise do código fonte do software, o corpo dos métodos não é verificado. A busca é em pré-ordem e percorre até encontrar os arquivos do projeto, pois a análise do conteúdo dos arquivos é feita pela ferramenta JCCD (Biegel; Diehl, 2010).

Após encontrar os arquivos do software, o *Plug-in AST Clone* os passa para o JCCD, detector de clones utilizando AST, para encontrar os trechos similares entre os arquivos. Estes trechos são separados em grupos de similaridade contendo informações de quais arquivos e respectivos trechos são clonados (*Trechos Clones*). Posteriormente, o *Plug-in AST Clone* faz o cálculo da métrica a fim de verificar quais blocos de código mais afetam no comportamento transversal do software. Para realizar o cálculo, o tamanho de cada clone é considerado como 40% do valor e o número de arquivos afetados tem relevância de 60% no valor final da métrica. Essa proporção é em decorrência da quantidade de arquivos afetados por um clone ser mais importante que o tamanho dos clones.

O *Plug-in AST Clone* utiliza o valor da métrica para acrescentar informações no editor de código por meio da criação de marcadores na barra lateral do editor Java. Um exemplo de marcadores utilizando o *Plug-in AST Clone* é apresentado na Figura 21.

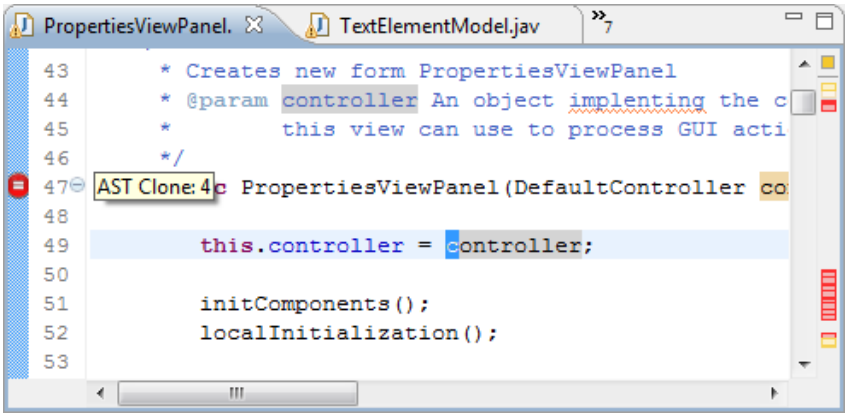


Figura 21 Marcadores do Plug-in AST Clone

O *Plug-in AST Clone* utiliza outros *Views*, além dos contidos no Eclipse para fornecer informações aos seus usuários. Um dos *Views* foi criado especialmente para o *Plug-in AST Clone* e outros dois são utilizados a partir de pontos de extensão do *plug-in* AJDT. O *View* criado é chamado *Clones View* e apresenta um conjunto de informações sobre (i) os grupos de arquivos clones, (ii) o tipo do bloco de código similar dentro do grupo de arquivos, (iii) o valor da métrica calculada, (iv) o número da linha inicial do trecho clonado e (v) o número da linha final do trecho clonado. O tipo do bloco é informado e segue o padrão de nomenclatura utilizada na implementação da ferramenta Antlr[7], a qual é utilizada internamente pelo JCCD. Um caso em que o *Clones View* foi utilizado para exibir os resultados da análise de um determinado projeto é apresentado na Figura 22.

[7] www.antlr.org

Para facilitar o processo de localização dos *Trechos Clones*, o *Plug-in AST Clone* permite ao usuário visualizar no editor Java o bloco de código clonado, semelhante à maneira como é feito para os *plug-ins* anteriormente citados. Uma visão mais geral sobre quanto os *Trechos Clones* afetam o sistema é fornecida utilizando o *View Visualizer* provido pelo *Plug-in AST Clone*. Como explicado em seções anteriores, o *Viusalizer*[8] permite análise universal sobre o código projeto.

Para utilizar o *Visualiser* no *Plug-in AST Clone*, *View* deve estar aberto e item que agrupa os blocos de clones do *Clones View* deve ser selecionado. Para o caso mostrado na Figura 23, ao selecionar o primeiro item da lista do *Clones View* ("Tipo de bloco: CONSTRUCTOR_DECL, valor da métrica: 4") marcadores de cor vermelha são criados no código fonte dos *Trechos Clones* que correspondem a este grupo, permitindo que sua exibição no *Visualiser*. Além de marcar os trecho selecionados, outros *Trechos Clones* são marcados em verde para o usuário verifique o quanto um determinado grupo de clones afeta o software.

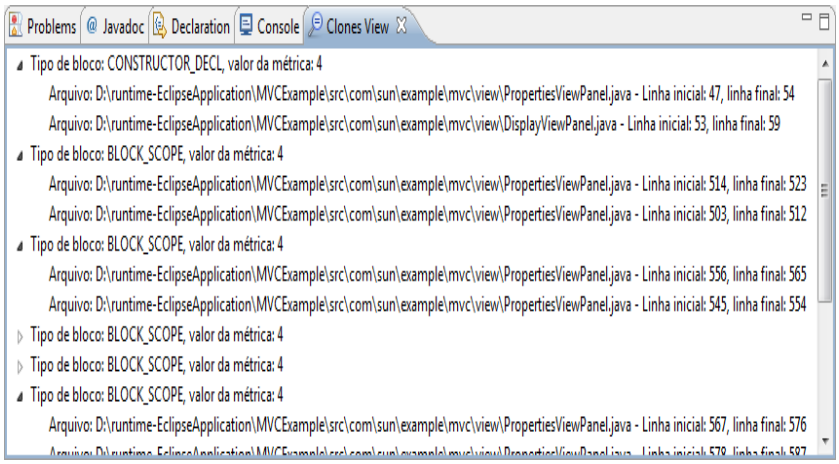


Figura 22 Clones View no Eclipse

[8] http://www.eclipse.org/ajdt/visualiser/

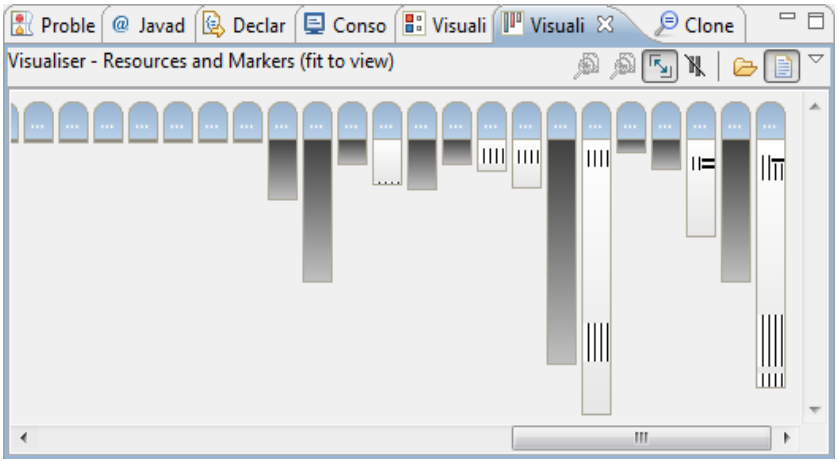


Figura 23 View Visualiser do AJDT utilizado pelo Plug-in AST Clone

Para facilitar a visualização, a cor verde foi substituida por linhas verticais e a cor vermelha por linhas horizontais. Tais modificações podem ser feitas a qualquer momento utilizando o *menu* de preferências do *Visualiser*. Como o *Visualiser* precisa carregar os arquivos do projeto para apresentar as informações, é necessário o usuário realizar o recarregamento, selecionando um projeto e voltando a carregar o projeto de seu interesse. Dessa forma, as alterações feitas nó código ao selecionar um item do *Clones View* passam a ser consideradas pelo *Visualiser*.

## 6.7. Considerações Finais

A construção de apoios computacionais para a detecção de interesses transversais não é uma atividade trivial, pois diversas características são imprescindíveis em sua concepção, tais como, facilidade de uso, compatibilidade com ambiente de desenvolvimento integrado, desempenho, acurácia e portabilidade. Além disso, deve cumprir com sua tarefa fundamental, encontrar trechos de código fonte que sejam indícios de interesses transverais.

Considerando esssas características, os *plug-ins* desenvolvidos foram implementados de forma independente, o que viabiliza melhoras futuras e permite acoplá-los a um conjunto mais extenso de ferramentas de mineração de aspectos. Além disso, o desenvolvimento separado de cada *plug-in* possibilita modularidade e manutenibilidade do seu código fonte.

# 7. ESTUDO DE CASO

## 7.1. Considerações Iniciais

Neste capítulo, é discutida a avaliação dos apoios computacionais desenvolvidos, realizada por meio de análises em softwares reais, e são brevemente comentadas algumas questões relacionadas ao desempenho. Durante a avaliação, foram conduzidos experimentos envolvendo o código fonte de softwares da categoria sistema de informação e de pequeno porte implementados na linguagem de programação Java: HealthWatcher (Soares *et al*., 2002), JCCD, MVCExample[9] e Praec.

Objetivos e métricas utilizadas para comparar técnicas e implementações são enunciados na Seção 7.2. Detalhes da condução do experimento são fornecidos na Seção 7.3. Resultados obtidos com a execução dos *plug-ins* no código fonte de softwares reais são apresentados na Seção 7.4. Ainda na Seção 7.4, são mencionados e apresentados as principais funções de cada software analisado, os resultados mais relevantes da execução de cada técnica em cada software analisado, tabelas comparativas com os resultados da execução e gráficos comparando o número de linhas e número de métodos dos softwares analisados com o tempo de execução das técnicas.

## 7.2. Objetivos e Métricas

A análise nos softwares orientados a objetos na linguagem de programação Java citados anteriormente têm o objetivo de verificar a eficiência dos *plug-ins* desenvolvidos e sua aplicabilidade em softwares reais. Além disso, é objetivo comparar as técnicas a fim de verificar qual delas aprensenta melhores

[9] http://java.sun.com/developer/technicalArticles/javase/mvc/MVCExample.zip

resultados para o usuário. Para os três *plug-ins* implementados foram analisados (i) a escalabilidade, (ii) a cobertura dos resultados e (iii) o tempo de execução.

A escalabilidade foi estudada a relação entre o crescimento do número de linhas e o crescimento do número de métodos, visto que duas das três técnicas implementadas oferecem como retorno os métodos do software. O tempo de execução foi medido a partir da introdução de instruções no código fonte para capturar o tempo antes e após a execução do código fonte referente à análise, desconsiderando o tempo gasto com apresentação dos resultados. A cobertura está relacionada com a qualidade dos resultados apontados pelas técnicas. No entanto, como os códigos fonte não foram analisados manualmente e não tinha especialista para verificar os resultados apresentados, a qualidade foi calculada baseando-se no resultados encontrados pelas três técnicas. Assim, se a técnica *A* não encontrou um interesse transversal identificado pelas outras duas técnicas, há indicação de que a técnica *A* falhou na cobertura desse interesse.

### 7.3. Condução do Experimento

Para comparar a complexidade dos softwares analisados, foram calculadas para cada software (i) a quantidade de classes, (iii) a quantidade de métodos e (iii) o LOC[10] utilizando o software LocMetrics[11].

Para capturar com fidelidade razoável a relação entre os tempos de execução, cada análise foi executada três vezes e a média entre os tempos foi calculada. Dessa forma, espera-se maior confiabilida nos valores obtidos, uma vez que diversos fatores podem influenciar a execução do software. Para análise Fan-In, foram comparados apenas os métodos declarado no softwares; assim,

[10] *Line of Code*
[11] http://www.locmetrics.com/

implementações não pertencentes aos softwares analisados foram desconsideradas nos resultados.

O equipamento utilizado para execução do experimento foi um computador modelo *Asus N61J*, com processador *Intel(R) Core(TM) i5 M520 2.4GHz*, 4 GB de memória RAM, e sistema operacional *Windows 7 Home Premium 64-bit*. Durante a execução, apenas o Eclipse e os processos básicos do sistema operacional estavam ativos.

### 7.4. Resultados Obtidos

O sistema MVCExample possui um total de 2350 linhas de código, 10 classes Java e 96 métodos. Foi desenvolvido exclusivamente para introduzir o conceito de desenvolvimento em camadas seguindo o modelo *Model-View-Controller*. O software possui a funcionalidade de um editor de texto no qual o usuário pode alterar a forma de exibição do conteúdo do editor, por exemplo, tamanho e cor da fonte, rotacionar o texto e alterar o tamanho da área disponível para entrada de dados.

Na execução do *Plug-in Fan-In*, dez métodos foram apontados como indícios de interesses transversais, o que corresponde a aproximadamente 10% do total de métodos do MVCExample. No entanto, sete destes não foram declarados pelos desenvolvedores, pois fazem parte de APIs externas utilizadas no desenvolvimento, e não serão considerados nos resultados. Dentre os três métodos identificados pelo *Plug-in Fan-In*, dois deles possuem chamadores da mesma classe, ou seja, são métodos da própria classe que invocam métodos identificados na análise: `setModelProperty()` da classe `AbstractController`, com 16 chamadores, e `createFontFromDialog()` da classe `JfontChooserDialog`, com 9

chamadores. O outro método detectado, `firePropertyChange()` da classe `AbstractModel`, possui 9 chamadores, mas de 2 classes distintas.

Para a execução do *Plug-in AST Clone*, os trechos de código encontrados pertencem principalemente a métodos de tratamento de eventos de interface e, em sua maioria, são da classe `PropertiesViewPanel`. Os *Trechos Clones* mais significativos encontrados estão presentes nas classes `PropertiesViewPanel` e `DisplayViewPanel` e têm, em média, 9 linhas semelhantes. Os *Trechos Clones* com média de 3 linhas de código foram encontrados nos métodos da classe `JFontChooserDialog`.

Assim como no resultado obtido com o *Plug-in Fan-In*, a análise utilizando o *Plug-in AST Clone* indica trechos de código da classe `JfontChooserDialog` como indícios de interesses transversais. No entanto, os resultados produzidos indicam métodos responsáveis por tratamento de eventos como possíveis interesses transversais e não o método `createFontFromDialog()` indicado pelo *Plug-in Fan-In*.

Os resultados obtidos com a utilização do *Plug-in Flow Graph* são mais próximos dos resultados obtidos pelo *Plug-in Fan-In*. O método com maior número de relações é o `createFontFromDialog()` da classe `JfontChooserDialog`, com 13 relações internas e 1 relação externa. As relações são com métodos da mesma classe. Outros métodos que merecem destaque são `firePropertyChange()` da classe `AbstractModel` e o método `setModelProperty()` da classe `AbstractController`, pois foram encontrados pelo *Plug-in Fan-In*. Os métodos encontrados pelo *Plug-in Flow Graph* correspondem a 38 % dos métodos do software, sendo que não foram aplicados filtros como no caso do *Plug-in Fan-In.*

Os principais resultados encontrados pelas técnicas são sumarizados na Tabela 5. Os campos marcados com “x” indicam que o método, disposto na linha da tabela, foi detectado pela técnica, disposto na coluna da tabela. Para melhor interpretação e comparação dos resultados produzidos pelas técnicas, essa tabela apresenta resultados em nível de métodos. No entanto, para a análise baseada no *Plug-in AST Clone*, os resultados podem ser observados em trechos de código.

Tabela 5 Sumarização dos Resultados da Análise no Software MVCExample

| Método | Fan-In | AST Clones | Flow Control |
|---|---|---|---|
| JFontChooserDialog.createFontFromDialog | x | | x |
| AbstractModel.firePropertyChange | x | | x |
| AbstractController.setModelProperty | x | | x |
| DisplayViewPanel.DisplayViewPanel | | x | |
| PropertiesViewPanel.PropertiesViewPanel | | x | |
| PropertiesViewPanel.widthTextFieldActionPerformed | | x | |

O software Praec foi desenvolvido como trabalho da disciplina de Modelagem e Implementação de Software ministrada aos alunos dos cursos de Ciência da Computação e de Sistemas de Informação da Universidade Federal de Lavras. Esse software consiste basicamente das funções de manutenção de cadastro de moradores e de residências, realização de pagamentos e geração de relatórios de pagamentos efetuados por moradores em um determinado período. O código do software é composto por 254 métodos, 26 classes e total de 5273 linhas de código.

As três análises foram executadas no código fonte desse software e resultados semelhantes aos do MVCExample foram encontrados. O *Plug-in Fan-In* e o *Plug-in Flow Control* encontraram indícios de interesses transversais em métodos relacionados com persistência de dados e comportamentos do modelo. Analisando o resultado por completo, é possível notar que o *Plug-in AST Clones* detectou com maior eficiência os trechos de códigos relacionados com tratamentos de eventos. Alguns métodos identificados pelos *plug-ins* são apresentados na Tabela 6, de forma semelhante como foram aprensentados os resultados na Tabela 5. Para o software Praec, o *Plug-in Fan-In* detectou interesse transversal em cerca de 5% do métodos do sistema, e o *Plug-in Flow Graph* em cerca de 22% dos métodos.

Outro software utilizado na avaliação dos *plug-ins* foi o HealthWatcher, comumente utilizado como um *benchmark* para estudos e experiementos relacionados as tecnologias de OO e OA em diferentes níveis do processo de desenvolvimento. O HealthWatcher é um software web que coleta e gerencia informações relacionadas a problemas do sistema de saúde. Os usuários podem acessar o software a partir de máquinas em locais públicos e realizar queixas ou pedir informações sobre o sistema de saúde. Quando da ocorrência de alguma queixa, esta é atribuida a um departamento específico e, assim que resolvida da maneira apropriada, a resposta deve ser encaminhada ao usuário que a efetuou. O código fonte do HealthWatcher é composto por 11644 linhas, distribuídas em 894 métodos e 135 classes.

Tabela 6 Sumarização dos Resultados da Análise no Software Praec

| **Método** | ***Fan-In*** | ***AST Clones*** | ***Flow Control*** |
|---|---|---|---|
| ConnectionFactory.connect | x | | x |

| | | | |
|---|---|---|---|
| ConnectionFactory.disconnect | x | | x |
| Morador.getMatricula | x | | |
| Apartamento.getNumero | x | | |
| TelaConsultarMorador.consultar | | x | |
| TelaRemoverMorador.consultar | | x | |
| ActionListener.actionPerformed | | x | |
| TelaEfetuarPagamento.btCancelarActionPerformed | | x | |
| Morador.getNome | | | x |
| MoradorDAO getInstance | | | x |

O código fonte do HealthWatch foi análisado pelos *plug-ins* e uma amostra dos resultados é apresentada na Tabela 7. É possível verificar a semelhança dos resultados encontrados pelo *Plug-in Fan-In* e pelo *Plug-in Flow Control*. *O Plug-in Fan-In* encontrou código transversal basicamente nas classes relacionadas a persistência de dados, por exemplo, `IpersistenceMechanism`; e classes da camada de controle, como é o caso da classe `ServletResponseAdapter`. O *Plug-In Flow Control* detectou interesse transversal na classe de persistência `IpersistenceMechanism`, em classes da camada de modelo do software, como é o caso `Complaint`, e em classes relacionadas a padrões de projeto, caso da classe `HealthWatcherFacade`. No o software HealthWatcher, o *Plug-in Fan-In* detectou interesse transversal em cerca de 3% do métodos do sistema, e o *Plug-in Flow Graph* em cerca de 25% dos métodos.

O JCCD, diferentemente dos softwares analisados anteriormente, não possui meios para interação com usuários finais, pois consiste de um conjunto de classes úteis para a realização de tarefas relacionadas a detecção de clones. Ele possui apenas uma interface de programação que pode ser utilizada por outros desenvolvedores para criar novas ferramentas. Resultados simplificados da

análise do código do JCCD pelos *plug-ins* são apresentados na Tabela 8. Uma vez que esse software não possui interesses trasnversais mais comuns encontrados na literatura, os resultados foram diferentes dos encontrados para os softwares anteriores. Neste caso, o *Plug-in Fan-In* e o *Plug-in Flow Control* encontraram indícios de interesses transversais em métodos de classes da camada de modelo, como é o caso da classe `ANode`. Assim como nos estudos anteriores, é bastante clara a similaridade dos resultados produzidos pelo *Plug-in Fan-In* e pelo *Plug-in Flow Control*. No o software JCCD, o *Plug-in Fan-In* detectou interesse transversal em cerca de 5% do métodos do sistema, e o *Plug-in Flow Graph* em cerca de 37% dos métodos.

Tabela 7 Sumarização dos Resultados da Análise no Software HealthWatcher

| **Método** | ***Fan-In*** | ***AST Clones*** | ***Flow Control*** |
|---|---|---|---|
| PersistenceMechanism.getCommunicationChannel | x | | |
| IPersistenceMechanism.getCommunicationChannel | x | | x |
| ServletResponseAdapter.getWriter | x | | x |
| CommandResponse.getWriter | x | | |
| ServletRequestAdapter.isAuthorized | x | | |
| HealthUnit.notifyObservers | | x | |
| Symptom.notifyObservers | | x | |
| ComplaintRepositoryArray.getIndex | | x | |
| SpecialityRepositoryArray.getIndex | | x | |
| SpecialityRepositoryArray.next | | x | |
| HealthUnitRepositoryArray.next | | x | |
| HealthWatcherFacade.getPm | | | x |
| Complaint.getCodigo | | | x |
| Complaint.getSituacao | | | x |
| PersistenceMechanism.getCommunicationChannel | x | | |

Tabela 8 Sumarização dos Resultados da Análise no Software JCCD

| **Método** | ***Fan-In*** | ***AST Clones*** | ***Flow Control*** |
|---|---|---|---|
| ANode.getType | x | | x |
| ANode.getChildCount | x | | x |
| ANode.getChild | x | | x |
| ASTManager.markChild | x | | |
| ASTManager.getChildCountOfMarkedNode | x | | x |
| CompareUnitThreading.checkNodeSimilarity | | x | |
| ASTCompareUnit.checkNodeSimilarity | | x | |
| JCCDEvaluation.countNodes | | x | |
| APipeline.countNodes | | x | |
| StandardPoolingStrategy.getPools | | x | |
| TokenLengthPoolingStrategy.getPools | | x | |
| ANode.getMarkedNode | | | x |
| ANode.getParent | | | x |

Os tempos de execução obtidos no experimento são descritos na Tabela 9. Cada linha da tabela corresponde a um software analisado. Para cada software, os *plug-ins* foram executados três vezes e os tempos obtidos em segundos estão dispostos nas colunas **T1**, **T2** e **T3**. A coluna **Média** corresponde à média aritmética dos três tempos. As colunas ***Fan-In***, ***AST Clones*** e ***Flow Control*** agrupam os resultados das três medições de cada *plug-in*.

Tabela 9 Tempo de Execução dos Plug-ins

| **Softwares** | **Tempo de Execução** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ***Fan-In*** | | | | ***AST Clones*** | | | | ***Flow Control*** | | | |
| | **T1** | **T2** | **T3** | **Média** | **T1** | **T2** | **T3** | **Média** | **T1** | **T2** | **T3** | **Média** |
| MVCExample | 19 | 17 | 17 | 17,7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Praec | 67 | 68 | 67 | 67,3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| HealthWatch | 225 | 221 | 240 | 228,7 | 4 | 1 | 1 | 2 | 4 | 1 | 1 | 2 |
| JCCD | 80 | 79 | 79 | 79,3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1,3 |

Para analisar a escalabilidade de cada técnica implementada, os tempos de execução foram obtidos e dois gráficos foram elaborados para melhor visualizar os resultados. Esses gráficos são apresentados na Figura 24 e na Figura 25, os quais mostram a relação entre a quantidade de métodos do software e quantidade de LOC com o tempo gasto na execução. Nessas figuras, os valores no eixo X foram gerados utilizando valores das métricas calculadas para os quatro softwares analisados. Ou seja, para a análise do *Tempo(s)* em função do *Número de Métodos*, os quatro softwares foram ordenados de acordo com a quantidade de métodos presentes no código e seus respectivos tempo de execução foram calculados. Em seguida, os gráficos foram montados para observar o comportamento de tempo de execução dos *plug-ins* em relação ao crescimento do número de métodos no software. Processo semelhante foi feito para obter um visão do tempo gasto na execução de acordo com o crescimento do número de linhas no código fonte.

Para ambos os gráficos, a linha azul marcada com losangos representa o tempo de execução do *Plug-in Fan-In*, a linha vermelha com retângulos o tempo de execução do *Plug-in AST Clone* e a linha verde com triângulos o tempo de execução do *Plug-in Flow Graph*. Analisando o gráfico da Figura 24, percebe-se que o tempo de execução tende a aumentar de acordo com o aumento do número de métodos do software, o que ocorre para os três *plug-ins*. Analisando o gráfico da Figura 25, outro comportamento é percebido quando é comparado o tempo de exeução e a quantidade de linhas de código fonte. Pode-se notar que nem sempre o software com mais linhas de código levará um tempo maior para ser analisado.

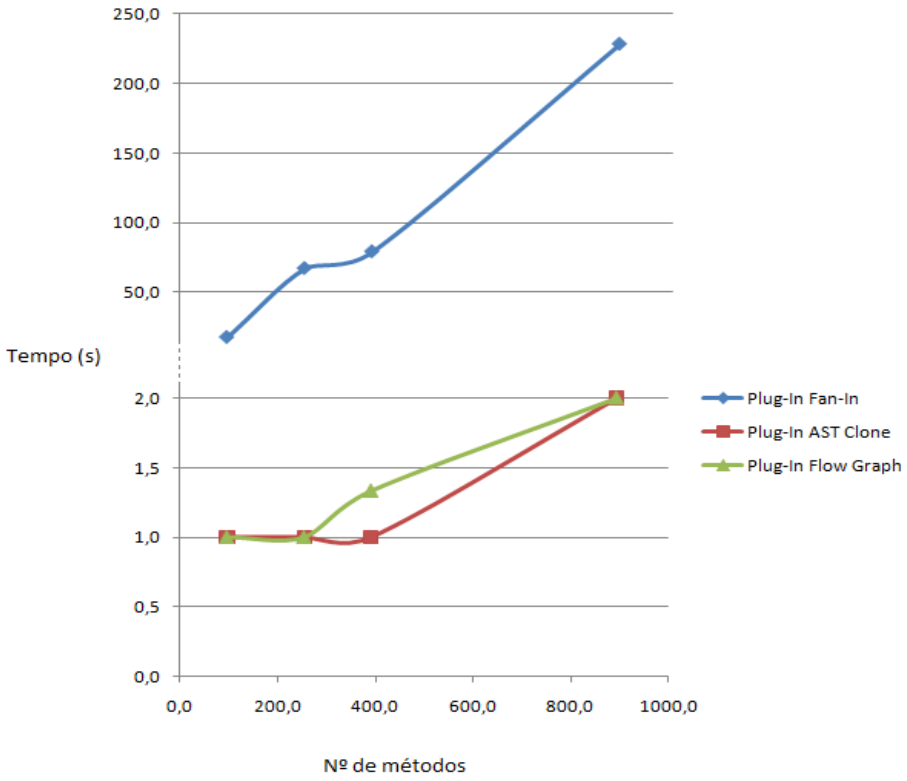


Figura 24 Número de Métodos x Tempo de Execução

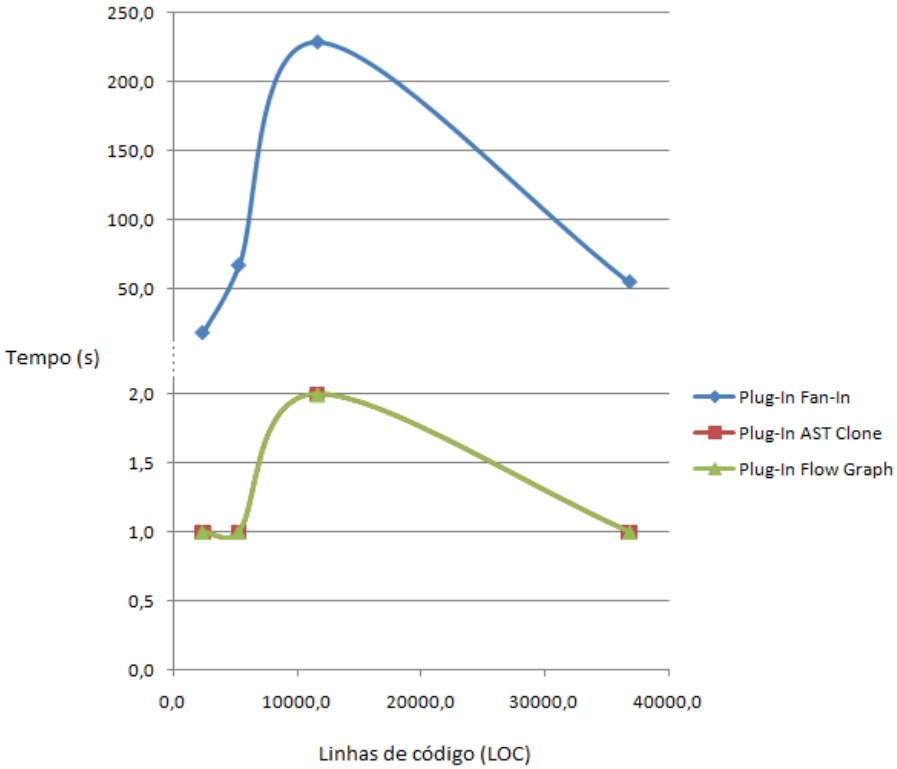


Figura 25 Número de LOC x Tempo de Execução

Em ambos os gráficos, percebe-se que o número de métodos do software tem maior influência no tempo de execução do que na quantidade de linhas no código fonte. Tal característica pode ser explicada pelo fato das técnicas implementadas nos *plug-ins* realizarem análise criteriosa sobre os métodos do software. Mesmo o *Plug-in AST Clone*, com granularidade em nível de trechos de código, realiza análises sobre o corpo de cada método do software.

## 7.5. Considerações Finais

Esta seção apresentou os principais resultados extraídos das análises efetuadas em quatro softwares da categoria sistemas de informação de pequeno porte, especificando as principais características de cada um deles e quais

trechos de código podem ser considerados como indícios de interesses transversais. Além disso, foi verificada a influência do número de linhas de código e número de métodos sobre o tempo de execução das técnicas implementadas.

Na comparação entre os *plug-ins*, percebeu-se proximidade dos resultados encontrados pelo *Plug-in Fan-In* e pelo *Plug-in Flow Control*. Dessa forma, uma boa abordagem é utilizá-los como complementares para detectar indícios de interesses transversais nos softwares.

No estudo do comportamento do tempo de execução com relação ao LOC e ao número de métodos, não se pode afirmar que o LOC é diretamente proporcional ao tempo, uma vez que o número de métodos do software exerce influência na execução das técnicas.

# 8. CONSIDERAÇÕES FINAIS

Interesses são implementações no código fonte responsáveis por tratar uma determinada função do software e, mesmo que este tenha sido bem modularizado em unidades como métodos e classes, algumas funções podem "atravessar" módulos comprometendo o encapsulamento. Uma possível forma de identificar interesses transversais em um software é utilizar mineração de aspectos do processo de refatoração orientada a aspectos, a qual permite representar interesses transversais identificados com mecanismos apropriados de linguagens de programação orientadas a aspectos.

Com o objetivo de aprimorar o processo de detecção de interesses transversais em um software, várias técnicas de análise de código estão presentes na literatura e algumas delas foram descritas nos capítulos anteriores. Entre as técnicas descritas, três delas foram escolhidas para serem implementadas em forma de *plug-in* para o Eclipse. As técnicas implementadas foram utilizadas para analisar o código fonte de softwares desenvolvidos na linguagem de programação Java e os resultados foram estudados a fim de comparar as técnicas quanto à eficiência e à qualidade.

Neste capítulo, são apresentadas algumas observações resultantes deste trabalho. Breve conclusão do trabalho é apresentada na Seção 8.1. Algumas contribuições obtidas com a realização deste trabalho são citadas na Seção 8.2. Alguns trabalhos futuros a serem desenvolvidos como desdobramento deste são sugeridos na Seção 8.3.

## 8.1. Conclusões

Após analisada a viabilidade de implementação de cada técnica, três delas foram escolhidas para serem codificadas como *plug-ins* para o Eclipse. Os

*plug-ins* foram desenvolvidos de forma independente prezando a modularização do código fonte visando a facilidade de futuras melhorias e foram testados para detectar indícios de interesses transversais em softwares reais. Durante os testes, critérios como cobertura e a relação entre tempo de execução e tamanho do software foram analisadas buscando verificar quais técnicas têm o melhor desempenho.

Com relação a tempo de execução, o *Plug-in AST Clone* e o *Plug-in Flow Graph* tiveram desempenho claramente superior ao *Plug-in Fan-In*. Por outro lado, o *Plug-in Fan-In* e o *Plug-in Flow Graph* mostraram melhores resultados quando avaliados no quesito cobertura dos resultados, uma vez que seus resultados foram semelhantes, além de detectar métodos relacionados com interesses transversais comumente conhecidos, neste caso, o de persistência.

Analisando os resultados obtidos, pode-se perceber que métodos relacionados a alguns interesses transversais no software foram identificados. Dentre os quais, destacam-se a persistência, tratamento de eventos, camada de controle e outros específicos da modelagem de cada software. Tais resultados evidenciaram o ganho de conhecimento sobre o código fonte dos softwares e mostraram como as técnicas de detecção de interesses transversais podem ser utilizadas para encontrar métodos e trechos de códigos com comportamento transversal no software.

## 8.2. Contribuições

Conforme apresentado neste trabalho, observam-se poucos casos na literatura relacionados a criação e a comparação de ferramentas para detecção de indícios de interesses transversais. Além disso, os escassos trabalhos relacionados foram desenvolvidos em instituições estrangeiras, o que deixa a literatura nacional praticamente fora deste cenário.

Desta forma, este trabalho deixa como contribuição um conjunto de ferramentas computacionais (*plug-ins*) para detecção de indícios de interesses transversais em softwares orientados a objetos escritos na linguagem de programação Java, além de um estudo acerca do comportamento das técnicas implementadas com relação ao tempo de execução da análise e sua variação de acordo com o tamanho do código fonte e número de métodos declarados no mesmo.

## 8.3. Trabalhos Futuros

A seguir, são apresentados alguns tópicos que podem ser levados em consideração no processo de continuidade deste trabalho:

- Implementar outras técnicas de detecção de interesses transversais visando criar um conjunto de ferramentas mais diverso e robusto. Além disso, uma ferramenta nacional com um grande número de abordagens de analise de código colocaria o Brasil em um ponto de maior destaque no cenário mundial de pesquisas em mineração de spectos;
- Aprimorar as ferramentas criadas com a funcionalidade de análise de dois ou mais projetos Java dependentes entre si, visto que grandes projetos são modularizados em diversos subprojetos Java no Eclipse. Esta funcionalidade possibilitaria a análise do código fonte de vários projetos e os resultados serem apresentados de umaforma combinada;
- Desenvolver um módulo para importar/exportar os resultados em forma de arquivos de *log*. Esta funcionalidade possibilitaria que análises de um mesmo software fossem comparadas durante vários estágios do desenvolvimento, uma vez que armazenados os resultados, os mesmos poderiam ser carregados pelos *plug-ins* em uma outra ocasião;
- Aperfeiçoar as ferramentas desenvolvidas acoplando os diversos *Views* necessários em uma única perspectiva. Isto daria ao usuário uma visão mais

clara de que os *plug-ins* fazem parte de um único ambiente de detecção de interesses transversais;

- Realizar estudos para encontrar novos métodos para detecção de indícios de interesses transversais. Uma boa abordagem é a incorporação de conceitos de outras áreas de pesquisa dentro da área de mineração de aspectos, tal como foi feito com os conceitos de detecção de clones, regras de associação e *clusters* em algumas das análises apresentadas neste trabalho.

## 9. REFERÊNCIAS BIBLIOGRÁFICAS


ANBALAGAN, P.; XIE, T. **Automated Inference of Pointcuts in Aspect-Oriented Refactoring**. ICSE '07: Proceedings of the 29th international conference on Software Engineering, Washington, DC, USA: IEEE Computer Society, 2007, pp. 127-136.

AOSDbr, C.B.D.D.D.S.O.A., "**Comunidade Brasileira de Desenvolvimento de Software Orientado a Aspectos para a Língua Portuguesa - AOSDbr**". Terminoloagia em português para orientação a aspectos. Brasília, 2007. Disponível em: <http://wiki.dcc.ufba.br/bin/view/AOSDbr/TermosEmPortugues>. Acessado em: Abr. 2010.

BANIASSAD, E.; CLEMENTS, P.C.; ARAUJO, J.; MOREIRA, A.; RASHID, A.; TEKINERDOGAN, B. **Discovering Early Aspects. Software**, IEEE, vol. 23, 2006, pp. 61-70.

BAXTER, I. D.; YAHIN, A.; MOURA, L.; SANT'ANNA, M.; Bier, L. **Clone Detection Using Abstract Syntax Trees. Software Maintenance**, 1998. Proceedings. International Conference on, 1998, pp. 368-377.

BHATTI, M. U.; DUCASSE, S. **Mining and Classification of Diverse Crosscutting Concerns**. LATE '08: Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution, New York, NY, USA: ACM, 2008, pp. 1-5.

BIEGEL, B.; DIEHL, S. **Highly Configurable and Extensible Code Clone Detection**. Proceedings of the 2010 17th Working Conference on Reverse Engineering, Washington, DC, USA: IEEE Computer Society, 2010, pp. 237-241.

BINKLEY, D.; CECCATO, M.; HARMAN, M.; RICCA, F.; TONELLA, P. **Automated Refactoring of Object Oriented Code into Aspects**. Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, 2005, pp. 27-36.

BREU, S.; KRINKE, J. **Aspect Mining Using Event Traces. Automated Software Engineering**, 2004. Proceedings. 19th International Conference on, 2004, pp. 310-315.

BRUNTINK, M**. Aspect Mining Using Clone Class Metrics**. 1st Workshop on Aspect Reverse Engineering, 2004 Disponível em:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.2557&rep=rep1&type=pdf>. Acessado: Mai. 2011.

BRUNTINK, M.; DEURSEN, A. van; TOURWE, T. **Isolating Idiomatic Crosscutting Concerns**. Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, 2005, pp. 37-46.

BRUNTINK, M.; DEURSEN, A. van; TOURWE, T.; ENGELEN, R. van. **An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns**. ICSM "04: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM"04), Washington, DC, USA: IEEE Computer Society, 2004, pp. 200-209.

BULL, R. I.; BEST, C.; STOREY, M. A. **Advanced Widgets for Eclipse**. Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, New York, NY, USA: ACM, 2004, pp. 6-11.

CECCATO, M.; MARIN, M.; MENS, K.; MOONEN, L.; TONELLA, P.; Tourwe, T. **A Qualitative Comparison of Three Aspect Mining Techniques**. IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension, Washington, DC, USA: IEEE Computer Society, 2005, pp. 13-22.

CHAPMAN, M. **Introducing AJDT: The AspectJ Development Tools**. International Business Machines Corporation, 2006. Disponível em: http://www.eclipse.org/articles/Article-Introducing-AJDT/article.html. Acessado em: Mar. 2011.

CHEN, S. C. **On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup**. IEEE Transactions on Computers, vol. 21, 1972, pp. 1293-1310.

COJOCAR, G. S.; SERBAN, G. **On Evaluating Aspect Mining Techniques**. Intelligent Computer Communication and Processing, 2007 IEEE International Conference on, 2007, pp. 217-224.

DIJKSTRA, E. W. **A Discipline of Programming**, Prentice Hall PTR, 1997.

FILMAN, R.; ELRAD, T.; CLARKE, S.; AKSIT, M. **Aspect-Oriented Software Development**. Addison-Wesley Professional, 2004.

GEER, D. **Eclipse Becomes the Dominant Java IDE**. Computer, v. 38, 2005, pp. 16-18.

GRISWOLD, W. G.; YUAN, J. J.; KATO, Y. **AspectBrowser for Eclipse**. 2005. Disponível em: <http://cseweb.ucsd.edu/~wgg/Software/AB/>. Acessado em: Mar. 2011.

GRISWOLD, W. G.; YUAN, J. J.; KATO, Y. **Exploiting the Map Metaphor in a Tool for Software Evolution**. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE '01. 2001. pp. 265 - 274.

HAN, J.; KAMBER, M. **Data Mining: Concepts and Techniques**. Morgan Kaufmann, 2000.

HANNEMANN, J.; KICZALES, G. **Overcoming the Prevalent Decomposition of Legacy Code**. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001). Workshop on Advanced Separation of Concerns in Software Engineering 2001.

HE, L.; BAI, H. **Aspect Mining Using Clustering and Association Rule Method**. International Journal of Computer Science and Network Security, 2006, pp. 247-251.

HENRY, S.; KAFURA, D. **Software Structure Metrics Based on Information Flow**. Software Engineering, IEEE Transactions on, vol. SE-7, 1981, pp. 510-518.

HORWITZ, S.; REPS, T. W. **The Use of Program Dependence Graphs in Software Engineering**. Proceedings of the 14th International Conference on Software Engineering, 1992, pp. 392-411.

IBM. **Eclipse Platform Technical Overview**, 2006. . Diponível em: <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>. Acessado em: Mar 2011.

IRWIN, J.; LOINGTIER, J. M.; GILBERT, J. R.; KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; SHPEISMAN, T. **Aspect-Oriented**

**Programming of Sparse Matrix Code**. ISCOPE ’97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments, London, UK: Springer-Verlag, 1997, pp. 249-256.

ISO/IEC 9126. ISO/IEC 9126-1 - **Software engineering** - Product quality - Part 1: Quality model (2001), ISO/IEC TR 9126-2 - Software engineering - Product quality - Part 2: External metrics (2003), ISO/IEC TR 9126-3 - Software engineering - Product quality - Part 3: Internal metrics (2003) e ISO/IEC TR 9126-4 - Software engineering - Product (2004).

JDT. **JDT Core Component**. The Eclipse Foundation, 2011. Disponível em: http://www.eclipse.org/jdt/core/. Acessado em: Mar. 2011.

JUNG, C.F. Metodologia para Pesquisa e Desenvolvimento**: Aplicada a Novas Tecnologias, Produtos e Processos**, Rio de Janeiro: Axcel Books do Brasil, 2004.

KAMIYA, T.; KUSUMOTO, S.; INOUE, K. CCFinder: **A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code**. Software Engineering, IEEE Transactions on, vol. 28, Jul. 2002, pp. 654-670.

KAWAKAMI, D. **Um Apoio Computacional para Auxiliar a Reengenharia de Sistemas Legados Java para AspectJ**. Dissertação de Mestrado. Universidade Federal de São Carlos, 2007.

KELLENS, A.; MENS, K.; TONELLA, P. **A Survey of Automated Code-Level Aspect Mining Techniques**. Transactions on Aspect-Oriented Software Development IV, A. Rashid and M. Aksit, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 143-162.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J. M.; IRWIN, J. **Aspect-Oriented Programming**. Europeen Conference on Object-Oriented Programming (ECOOP), M.A.A.S. Matsuoka, ed., Springer Verlag, 1997, pp. 220-242.

KOMONDOOR, R.; HORWITZ, S. **Using Slicing to Identify Duplication in Source Code**. SAS ’01: Proceedings of the 8th International Symposium on Static Analysis, London, UK: Springer-Verlag, 2001, pp. 40-56.

KRINKE, J. **Identifying Similar Code with Program Dependence Graphs**. In Proc. Eigth Working Conference on Reverse Engineering, 2001, pp. 301-309.

KRINKE, J.; BREU, S. **Control-Flow-Graph-Based Aspect Mining**. 1st Workshop on Aspect Reverse Engineering, Citeseer, 2004, pp. 1-5.

KUCK, D. J.; MURAOKA, Y.; CHEN, S. C. **On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup**. IEEE Transactions on Computers, vol. 21, 1972, pp. 1293-1310.

LADDAD, R. **Aspect Oriented Refactoring**. Addison-Wesley Professional, 2006.

LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**, Greenwich, CT, USA: Manning Publications Co., 2003.

LAKATOS, E.M.; MARCONI, M.D. **Fundamentos De Metodologia Científica**, São Paulo: 2001.

LOPES, C. V. **D: A Language Framework For Distributed Programming**. College of Computer Science of Northeastern University, 1997. Disponível em: <http://www2.parc.com/csl/groups/sda/publications/papers/Lopes-Thesis/Bibliography.pdf>. Acessado em: Mar 2011.

MARIN, M. **FINT - Tool Support for Aspect Mining**. 2008. Disponível em: http://swerl.tudelft.nl/bin/view/AMR/FINT. Acessado em Mar 2011.

MARIN, M.; DEURSEN, A. V.; MOONEN, L. **Identifying Crosscutting Concerns Using Fan-In Analysis**. ACM Trans. Softw. Eng. Methodol., vol. 17, 2007, pp. 1-37.

MARIN, M.; DEURSEN, A. van; MOONEN, L. **Identifying Aspects Using Fan-In Analysis**. Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004, pp. 132-141.

MARIN, M.; MOONEN, L.; DEURSEN, A. van. **FINT: Tool Support for Aspect Mining**. Proceedings of the 13th Working Conference on Reverse Engineering, Washington, DC, USA: IEEE Computer Society, 2006, pp. 299-300.

MENDHEKAR, A.; KICZALES, G.; Lamping, J. **RG: A Case-Study for Aspectoriented Programming**. 1997.

NOVAIS, R. L. **Uma Arquitetura para Ferramentas de Aspect Mining**. Bahia, 2009. Disponível em: <http://im.ufba.br/MATA23/TrabalhoRenatoNovais>. Acessado em Mar. 2010.

Pawlak, R.; RETAILLÉ, J. P.; SEINTURIER, L. **Foundations of AOP for J2EE Development (Foundation)**, Berkely, CA, USA: Apress, 2005.

PFLEEGER, S. L.; ATLEE, J. M. **Software Engineering: Theory and Practice**. Prentice Hall. 792 pages. 2009.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. McGraw-Hill. 928 pages. 2009.

QU, L.; LIU, D. **Aspect Mining Using Method Call Tree**. Multimedia and Ubiquitous Engineering, 2007. MUE ’07. International Conference on, 2007, pp. 407-412.

RESENDE, A. M. P. de; SILVA, C. C. da. **Programação Orientada a Aspectos em Java - Desenvolvimento de Software Orientado a Aspectos**, Rio de Janeiro: Brasport, 2005.

ROBILLARD, M. P.; MURPHY, G. C. **Representing Concerns in Source Code**. ACM Trans. Softw. Eng. Methodol., vol. 16, 2007.

ROY, C. K.; UDDIN, M. G.; ROY, B.; DEAN, T. R. **Evaluating Aspect Mining Techniques: A Case Study**. Program Comprehension, 2007. ICPC ’07. 15th IEEE International Conference on, 2007, pp. 167-176.

SHEPHERD, D.; GIBSON, E.; POLLOCK, L. L. **Design and Evaluation of an Automated Aspect Mining Tool**. Software Engineering Research and Practice, H.R. Arabnia and H. Reza, eds., CSREA Press, 2004, pp. 601-607.

SHEPHERD, D.; PALM, J.; POLLOCK, L.; CHU-CARROLL, M. TIMNA: **A Framework for Automatically Combining Aspect Mining Analyses**. ASE ’05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA: ACM, 2005, pp. 184-193.

SOARES, S., LAUREANO, E., and BORBA, P. **Implementing distribution and persistence aspects with AspectJ**" In ACM Press, editor, 17th ACM conference OOPSLA'02, pages 174 –190, 2002.

SOMMERVILLE, I. **Software Engineering** (International Computer Science Series), Addison Wesley, 2010.

TAKASHIO, K.; TOKORO, M. DROL: **An Object-Oriented Programming Language for Distributed Real-Time Systems**. OOPSLA, 1992, pp. 276-294.

TARR, P.; OSSHER, H.; HARRISON, W.; SUTTON Jr., S. M. N **Degrees of Separation: Multi-Dimensional Separation of Concerns**. Software Engineering, 1999. Proceedings of the 1999 International Conference on, 1999, pp. 107-119.

TOURWE, T.; MENS, K. **Mining Aspectual Views Using Formal Concept Analysis**. Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on, 2004, pp. 97-106.

UML. **Unified Modeling Language**. Disponível em: <http://www.uml.org/>. Acessado em: Mai 2011.

VALLÉE-RAI, R.; CO, P.; GAGNON, E.; HENDREN, L.; LAM, P.; SUNDARESAN, V. **Soot - A Java Bytecode Optimization Framework**. 1999.

YOSHIKIYO, W. G.; KATO, Y.; YUAN, J. J. **Aspect Browser: Tool Support for Managing Dispersed Aspects**. In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems - OOPSLA 99, 1999.

YUEN, I.; ROBILLARD, M. P. **Bridging the Gap Between Aspect Mining and Refactoring**. LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution, New York, NY, USA: ACM, 2007, p. 1.

ZHANG, C.; JACOBSEN, H. A. **Efficiently Mining Crosscutting Concerns Through Random Walks**. AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, New York, NY, USA: ACM, 2007, pp. 226-238.