UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM METAMODELO PARA FACILITAR A INTEGRAÇÃO DE FERRAMENTAS DE VISUALIZAÇÃO DE SOFTWARE E MINERAÇÃO DE INTERESSES TRANSVERSAIS

OSCAR JOSÉ FERNANDES TANNER

ORIENTADORA: PROFA. DRA. ROSÂNGELA AP. DELLOSSO PENTEADO

São Carlos – SP

Outubro/2013

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM METAMODELO PARA FACILITAR A INTEGRAÇÃO DE FERRAMENTAS DE VISUALIZAÇÃO DE SOFTWARE E MINERAÇÃO DE INTERESSES TRANSVERSAIS

OSCAR JOSÉ FERNANDES TANNER

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software

Orientadora: Profa. Dra. Rosângela Ap. Dellosso Penteado

São Carlos – SP

Outubro/2013

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

T167mf

Tanner, Oscar José Fernandes.
Um metamodelo para facilitar a integração de ferramentas de visualização de software e mineração de interesses transversais / Oscar José Fernandes Tanner. -- São Carlos : UFSCar, 2014.
128 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2013.

1. Engenharia de software. 2. Orientação a aspectos. 3. Interesses transversais. 4. Mineração de interesses transversais. 5. Visualização de software. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Um Metamodelo para Facilitar a Integração de Ferramentas de Visualização de Software e Mineração de Interesses Transversais"

Oscar José Fernandes Tanner

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Profa. Dra. Rosângela Ap. Dellosso Penteado
(Orientadora - DC/UFSCar)

Prof. Dr. Valter Vieira de Camargo
(DC/UFSCar)

Prof. Dr. Glauco de Figueiredo Carneiro
(UNIFACS)

São Carlos
Novembro/2013

Este trabalho é dedicado a minha querida e amada Oma.

AGRADECIMENTOS

À minha família, por sempre me apoiar e acreditar em meu potencial.

À Carol por todo apoio e paciência.

À Professora Dra. Rosângela pela amizade, confiança e orientação.

Ao companheiro de laboratório Matheus, pela amizade e horas gastas me ajudando com minhas dúvidas.

Ao Professor Dr. Glauco Carneiro e ao Arleson Silva por terem disponibilizado a SourceMiner para a realização deste trabalho.

Ao Professor Dr. Rogério Garcia e a Fernanda Delfim por realizarem a integração da $SoftVis_{4CA}$

Ao Sr. Hung Viet Nguyen por disponibilizar o código fonte da ferramenta XScan para a realização deste trabalho.

RESUMO

Um dos objetivos do desenvolvimento Orientado a Objetos é a construção de software com melhor modularização e separação de interesses, porém não há como evitar que haja entrelaçamento e espalhamento de alguns desses interesses. O desenvolvimento Orientado a Aspecto é uma proposta de solucionar esse problema. Entretanto, a identificação de interesses entrelaçados e espalhados pelo código fonte, conhecidos como interesses transversais (IT), não é simples. Para auxiliar essa identificação é recomendado o uso de ferramentas de mineração de interesses transversais (MIT). Após esses interesses serem identificados, muitas vezes, persiste a dificuldade de visualizá-los adequadamente no código fonte. Uma forma de melhorar a apresentação dos ITs é exibir os resultados das ferramentas de MIT usando as ferramentas de visualização de software. A visualização de software tem como objetivo abstrair as informações estruturais de um software por meio de metáforas visuais. Porém, nem sempre é possível integrar facilmente essas ferramentas, pois cada uma armazena seus resultados em um formato específico que normalmente não é reconhecida pelas demais. Para solucionar esse problema, neste trabalho foi criado um metamodelo que facilita essa integração. Os resultados obtidos a partir de uma ferramenta de MIT são transformados para o formato desse metamodelo, para que sejam processados por uma ferramenta de visualização de software. A exibição dos resultados das ferramentas de MIT por meio das ferramentas de visualização facilita a comparação desses resultados, sendo evidenciados os falsos negativos e facilitada a identificação de novos padrões visuais para falsos positivos. Para mostrar a utilidade de um ambiente integrado, foram utilizadas quatro ferramentas de MIT e duas de visualização de software. Além disso, dois experimentos foram conduzidos: um para avaliar a utilização do metamodelo proposto, e outro para avaliar a utilização do ambiente integrado. Os resultados apontaram que a integração é facilitada com a utilização do metamodelo proposto e que é necessário que o engenheiro de software tenha pleno domínio de cada uma das ferramentas envolvidas, a fim de beneficiar-se dessa integração.

Palavras-chave: Orientação a Aspectos, Interesses Transversais, Mineração de Interesses Transversais, Visualização de Software.

ABSTRACT

A goal of the Object-Oriented Programming is to develop software with better modularization and separation of concerns, although tangling and scattering of some of these concerns cannot be prevented. The Aspect-Oriented Programming was proposed to solve this problem. However, the identification of the scattering and tangling concerns, also known as crosscutting concerns (CC), is not simple. To assist this identification, it is recommended the use of CC Mining tools (MT), but some MTs do not properly display these CCs in the source code. One way to improve the presentation of this CCs is displaying them through software visualization tools. Software visualization aims to abstract the structural information of the software through visual metaphors. However, it is not always simple to integrate CC and visualization tools, because each tool store their results in a specific format that is usually not recognized by the others. In order to solve this problem, this work created a metamodel that facilitates this integration. The results obtained from a MT are transformed to this metamodel format, so they can be processed by a software visualization tool. Presenting the results of a MT through software visualization tools helps the comparison of these results, facilitating the detection of false negatives and the identification of visual patterns that indicate false positives. Four MT and two software visualization tools were integrated to demonstrate the benefits of an integrated environment. Moreover, two experiments were conducted: one to evaluate the use of the proposed metamodel; and another to evaluate the use of the integrated environment. The results revealed that the use of the proposed metamodel facilitates this integration and the software engineer must be a specialist of each of the integrated tools in order to enjoy the benefits from this integration.

Keywords: Aspect-Oriented Programming, Crosscutting Concerns, Crosscutting Concern Mining, Software Visualization

SUMÁRIO

CAPíTULO 1 – INTRODUÇÃO **1**

1.1 Contexto 1

1.2 Motivação e Objetivos 4

1.3 Metodologia de Desenvolvimento do Trabalho 6

1.4 Trabalhos relacionados 7

1.5 Organização do Trabalho 8

CAPíTULO 2 – ORIENTAÇÃO A ASPECTOS E MINERAÇÃO DE INTERESSES TRANSVERSAIS **9**

2.1 Considerações Iniciais 9

2.2 Programação Orientada a Aspectos 10

2.3 Mineração de Interesses Transversais 11

2.3.1 Técnicas de Mineração de Interesses Transversais 11

2.3.1.1 Análise Exploratória 11

2.3.1.2 Análise de *Fan-in* 12

2.3.1.3 Análise por Detecção de Clones 13

2.3.1.4 *Concern Peers* 14

2.3.1.5 Tipos e Texto 14

2.3.2 Ferramentas de Mineração de Interesses Transversais 15

2.3.2.1 FEAT 16

2.3.2.2 ConcernMapper 17

2.3.2.3 FINT 17
2.3.2.4 CCFinder 18
2.3.2.5 XScan 19
2.3.2.6 ComSCId 20
2.4 Considerações Finais 22

CAPíTULO 3 – VISUALIZAÇÃO DE SOFTWARE **24**

3.1 Considerações Iniciais 24
3.2 Conceitos de Visualização de Software 25
3.3 Ferramentas de Visualização de Software 26
3.3.1 MosaiCode 28
3.3.2 PEVAD 29
3.3.3 SourceMiner 30
3.3.4 $SoftVis_{4CA}$ 35
3.3.5 VizzAspectJ-3D 37
3.4 Considerações Finais 38

CAPíTULO 4 – METAMODELO PROPOSTO **40**

4.1 Considerações Iniciais 40
4.2 Criação do Metamodelo 42
4.3 Metamodelo Programado 45
4.3.1 Pacote `model` 46
4.3.2 Pacote `writers` e `helper` 47
4.3.2.1 Classe `IMMWriter` 48
4.3.2.2 Classe `CMWriter` 48
4.3.2.3 Classe `CMTypeFormatConverter` 50
4.3.3 Pacotes `readers` e `readers.model` 50

4.4 Considerações Finais 50

CAPíTULO 5 – UMA UTILIZAÇÃO DO METAMODELO INDICATIONS METAMODEL 52

5.1 Considerações Iniciais 52

5.2 Procedimentos de Integração 54

5.2.0.1 ComSCId 55

5.2.0.2 XScan 57

5.2.0.3 ConcernMapper 59

5.2.0.4 FINT 61

5.2.0.5 $SoftVis_{4CA}$ 62

5.2.0.6 SourceMiner 64

5.3 Prova de Conceito 66

5.4 Considerações Finais 77

CAPíTULO 6 – EXPERIMENTOS SOBRE A INTEGRAÇÃO E A UTILIZAÇÃO DAS FERRAMENTAS MIT E DE VISUALIZAÇÃO DE SOFTWARE 80

6.1 Considerações Iniciais 80

6.2 Experimento de Utilização do IMM 81

6.2.1 Definição 81

6.2.1.1 Objetos de Estudo 81

6.2.1.2 Objetivo 82

6.2.1.3 Enfoque Quantitativo 82

6.2.1.4 Perspectiva 82

6.2.1.5 Contexto 82

6.2.2 Planejamento 82

6.2.2.1 Seleção do contexto 83

6.2.2.2 Formulação das Hipóteses 83

6.2.2.3 Seleção das Variáveis 84
6.2.2.4 Seleção dos Participantes 84
6.2.2.5 Projeto do Experimento 84
6.2.2.6 Projeto dos Tipos 84
6.2.2.7 Instrumentação 85
6.2.3 Operação 85
6.2.3.1 Preparação 85
6.2.3.2 Execução 85
6.2.4 Análise dos Dados 86
6.2.4.1 Estatísticas Descritivas 86
6.2.4.2 Testando as Hipóteses 88
6.2.5 Ameaças à Validade 90
6.3 Experimento de Utilização do Ambiente Integrado 91
6.3.1 Definição 91
6.3.1.1 Objetos de Estudo 92
6.3.1.2 Objetivo 92
6.3.1.3 Enfoque Quantitativo 92
6.3.1.4 Enfoque Qualitativo 92
6.3.1.5 Perspectiva 93
6.3.1.6 Contexto 93
6.3.2 Planejamento 93
6.3.2.1 Seleção do contexto 93
6.3.2.2 Formulação das Hipóteses 94
6.3.2.3 Seleção das Variáveis 95
6.3.2.4 Seleção dos Participantes 95
6.3.2.5 Projeto do Experimento 96

6.3.2.6 Projeto dos Tipos 96
6.3.2.7 Instrumentação 96
6.3.3 Operação 97
6.3.3.1 Preparação 97
6.3.3.2 Execução 97
6.3.4 Análise dos Dados 98
6.3.4.1 *Processer* e *Splitter* 98
6.3.4.2 Estatísticas Descritivas 99
6.3.4.3 Testando as Hipóteses 101
6.3.5 Ameaças à Validade 103
6.3.6 Opinião dos participantes quanto à utilização do ambiente integrado de ferramentas de MIT e de visualização de software 105
6.3.7 Avaliação da Usabilidade das Visualizações da SourceMiner 107
6.4 Considerações Finais 108

CAPíTULO 7 – CONCLUSÕES **110**

7.1 Contribuições deste Trabalho 111
7.2 Limitações deste Trabalho 113
7.3 Sugestões de Trabalhos Futuros 113

REFERÊNCIAS **115**

GLOSSÁRIO **121**

APÊNDICE A – FORMULÁRIO DE EXECUÇÃO DO EXPERIMENTO DE INTEGRAÇÃO **122**

APÊNDICE B – DESCRITIVO DO FORMATO CM **123**

APÊNDICE C – FORMULÁRIO DE CONSENTIMENTO **125**

LISTA DE FIGURAS

1.1 Esquema de uma arquitetura integrada de ferramentas de MIT e de visualização de software. 5

2.1 Tela da ferramenta FEAT. 16

2.2 Exemplo de utilização da ConcernMapper. 18

2.3 Interface de filtros da ferramenta FINT. 18

2.4 Exemplo de resultados exibidos pela ferramenta FINT. 19

2.5 Interface para cadastrar, alterar e remover regras. 21

2.6 Apresentação dos indícios de persistência na visão de árvore. 22

3.1 Exemplo de visualização sem filtro. 27

3.2 Exemplo de visão com filtro. 27

3.3 Tela da ferramenta MosaiCode, extraída de (MALETIC *et al.*, 2011). 29

3.4 Exemplo da Visão *Treemap*. 30

3.5 Exemplo da Visão Polimétrica. 31

3.6 Exemplo da Visão de Dependência. 32

3.7 Exemplo da Visão de Acoplamento. 32

3.8 Exemplo da Visão Espiral. 33

3.9 Exemplo da Visão Matriz de Acoplamento. 33

3.10 Exemplo da Visão *Treemap* colorida de acordo com alguns indícios do IT de persistência. 34

3.11 Exemplo da Visão Inter-Classes, extraído de (DELFIM, 2013). 35

3.12 Exemplo da Visão Inter-Métodos, extraído de (DELFIM, 2013). 35

3.13 Exemplos das Visões Inter-Instruções, extraídos de (DELFIM, 2013). 36
3.14 Exemplo da *Visão Estrutural*, extraído de (DELFIM, 2013). 37
3.15 Exemplo da *Visão de Distribuição de Instruções*, extraído de (DELFIM, 2013). 37
3.16 Exemplo da *Visão de Bytecode*, extraído de (DELFIM, 2013). 37
3.17 Tela da ferramenta VizzAspectJ-3D, extraída de (BENTRAD; MESLATI, 2011). 38
4.1 *Indications Metamodel*. 44
4.2 Estrutura das classes do metamodelo implementado em linguagem Java. 46
5.1 Classe Package decorada. 56
5.2 Menu para criação do arquivo IMM com os indícios de ITs identificados pela ComSCId. 57
5.3 XScan, execução iniciada por menu principal. 58
5.4 XScan, execução por item de menu acessado ao clicar com botão direito do mouse sobre o projeto a ser analisado. 58
5.5 Arquivos IMM gerados pela XScan na raiz do projeto analisado. 60
5.6 Menu para converter um arquivo CM para o formato IMM. 60
5.7 Arquivo IMM criado a partir do arquivo CM. 61
5.8 Item de menu para converter o arquivo da ferramenta FINT para o formato IMM. 61
5.9 Exemplo da integração com a ferramenta $SoftVis_{4CA}$. 63
5.10 Item de menu para converter qualquer arquivo IMM em um arquivo CM. 65
5.11 Perspectiva SourceMiner. 66
5.12 Perspectiva SourceMiner (FINT). 66
5.13 Indícios encontrados na primeira execução da ferramenta ComSCId. 68
5.14 Exemplos de métodos presentes na área C da Figura 5.13 69
5.15 Indícios encontrados após remover a classe java.sql.Date das regras da ComSCId. 69
5.16 Método `getToday()`. 70
5.17 Indícios identificados pela ComSCId após a remoção da regra que identificava a função getToday(). 70

5.18 Cabeçalho da classe `RowNotFoundException`. 71
5.19 Resultado obtido pela mineração da XScan. 72
5.20 Indícios encontrados pela ComSCId e pela XScan combinados pela utilização de uma mesma cor para representá-los. 73
5.21 Indícios encontrados pela ComSCId e pela XScan com cores diferentes para possibilitar a sua comparação. 73
5.22 Abas da ferramenta ConcernMapper. 74
5.23 Descrição de um dos métodos do grupo `#6` identificados pela XScan. 75
5.24 Resultado obtido após a criação de regra para identificar todos os métodos cujos nomes contém o conjunto de caracteres “Clause”. 75
5.25 Aba da ferramenta FINT com os métodos identificados. 76
5.26 Resultado final da mineração do IT de persistência. 76
5.27 Aba da ferramenta FINT com indícios do IT de tratamento de exceção destacados. 77
5.28 Indícios do IT de tratamento de exceção identificados pela ferramenta FINT . . 78
6.1 Boxplots dos dados da Tabela 6.1. 88
6.2 Testes de normalidade dos dados da Tabela 6.1. 89
6.3 Testes de Inferência 90
6.4 Boxplots dos dados da Tabela 6.2. 101
6.5 Testes de normalidade dos dados da Tabela 6.3. Parte 1. 102
6.6 Testes de normalidade dos dados da Tabela 6.3. Parte 2. 103
6.7 Testes de Inferência. 104
6.8 Questionário de avaliação do ambiente integrado. 106
6.9 Gráfico com a opinião dos participantes. 106
6.10 Resultados do questionário de usabilidade das versões da SourceMiner. 107
6.11 Questionário de usabilidade das versões da SourceMiner. 108

LISTA DE TABELAS

2.1 Sumarização dos pontos positivos e negativos das Técnicas de MIT. 23

3.1 Número de Visões e Metáforas das ferramentas apresentadas. 39

4.1 Sumarização da conversão de tipos do formato Java para o formato CM. 49

6.1 Dados coletados no experimento de integração usando os formatos IMM e CM. 87

6.2 Dados coletados no experimento de utilização do ambiente integrado. 100

6.3 Dados coletados no experimento de utilização do ambiente integrado após retirada dos *outliers*. 100

Capítulo 1

INTRODUÇÃO

1.1 Contexto

O desenvolvimento de software nem sempre segue as etapas preconizadas por um dos modelos de processo. Na maioria das vezes, somente o código fonte está disponível para o cliente e para os desenvolvedores (*stakeholders*). Dessa forma, a atividade de manutenção fica prejudicada consumindo, em média, 90% do tempo das atividades realizadas nas empresas que utilizam e desenvolvem software (PRESSMAN, 2001).

Os desenvolvedores devem cuidar para que os sistemas sejam gerados especificando tanto os requisitos funcionais como os requisitos não funcionais. Muitas vezes, por falta de conhecimento do desenvolvedor ou por prazos inapropriados colocados pelos clientes, técnicas consolidadas de Engenharia de Software não são utilizadas, o que leva, geralmente, a produtos com qualidade questionável. Na maioria dos casos os desenvolvedores atentam para as necessidades imediatas do cliente, esquecendo-se que esse software irá passar por atividades de manutenção. Quando essas atividades são realizadas surgem as dificuldades tanto para inclusão quanto para correção de uma ou mais funções que o sistema necessite.

A Orientação a Objetos (OO) (PRESSMAN, 2001) foi uma tentativa de resolver esses problemas, proporcionando facilidade no desenvolvimento, gerando softwares mais reusáveis e modularizados e reduzindo o custo de manutenção. Embora fosse preconizado que sistemas desenvolvidos com OO seriam mais modularizados, não há como evitar o espalhamento e o entrelaçamento de interesses nos diversos módulos e componentes do software (KICZALES *et al.*, 2001). Um interesse é qualquer requisito que o cliente considera como uma unidade conceitual, incluindo as regras de negócio, os requisitos funcionais e não funcionais do sistema (ROBILLARD; MURPHY, 2007). Um interesse espalhado e entrelaçado, por entrecortar trans-

versalmente as demais estruturas de um sistema, é também chamado de Interesse Transversal (IT) (KICZALES *et al.*, 1997).

Como uma proposta de complementar a OO e melhorar o cenário descrito, surge a Programação Orientada a Aspectos (POA) (KICZALES *et al.*, 1997), fornecendo estruturas capazes de encapsular esses ITs em unidades modulares chamadas de aspectos. Exemplos de ITs são persistência em banco de dados, *logging*, concorrência, autenticação de usuário e criptografia. Por meio da POA, os sistemas podem ser melhor modularizados, aumentando sua manutenibilidade. Porém, identificar esses ITs não é trivial e necessita de ferramentas de apoio que auxiliem e guiem o engenheiro de software durante esse processo, como as ferramentas de mineração de interesses transversais. Porém, é interessante que esses ITs após serem identificados sejam exibidos para que o engenheiro de software possa localizá-los no código fonte. As ferramentas de visualização de software é que dão apoio a essa tarefa.

A Mineração de Interesses Transversais (MIT) tem como objetivo identificar trechos do código fonte que implementem os requisitos de algum IT (ZHANG; JACOBSEN, 2012). As atividades a serem realizadas durante um processo de MIT estão associadas a uma ou mais técnicas de mineração. Cada técnica ou conjunto de técnicas, possui pelo menos uma ferramenta que a implemente.

Existem diversas técnicas de MIT presentes na literatura, dentre elas:

- **Análise Exploratória** (ROBILLARD; MURPHY, 2002) é responsabilidade do engenheiro de software explorar todas as estruturas de um sistema procurando pelos indícios de ITs. Dessa forma, o engenheiro de software deve ter total conhecimento de todos os ITs para identificá-los corretamente. Essa técnica é suscetível a erros, além de demandar grande esforço.

- **Análise de *Fan-in*** (MARIN *et al.*, 2007) utiliza o conceito de que métodos espalhados e entrelaçados pelo software possuem um alto valor de acordo com a métrica Fan-in. Dessa forma esses métodos são sugeridos ao engenheiro de software como indícios de ITs, cabendo a ele analisá-los a fim de determinar se realmente são indícios e qual o IT que representam.

- **Tipos e texto** (HANNEMANN; KICZALES, 2001) utiliza regras para identificar indícios por meio de seus tipos (classes) e por meio de um conjunto de caracteres utilizado para procurar por nome de classes, métodos, variáveis e valores atribuídos a variáveis. Essa técnica reduz a necessidade de conhecimento prévio do engenheiro de software e do IT

procurado, já que um mesmo conjunto de regras pode ser utilizado para analisar diversos softwares.

- ***Concern Peers*** (NGUYEN *et al.*, 2011) é totalmente automatizada, quando a ferramenta é usada não é possível que o engenheiro de software faça modificações ou refinamentos no conjunto de resposta que foi obtido a partir do uso da ferramenta.

Em algumas situações pode ser interessante ter o uso combinado de técnicas e ferramentas. Ceccato *et al.* (2006) analisaram a combinação de três técnicas diferentes de MIT para identificar os indícios de ITs presentes no sistema JHotDraw (JHOTDRAW.ORG, 2013), um *framework* de editor gráfico *open source*. Ao analisarem os resultados obtidos no fim do processo de mineração, os autores perceberam que essas três técnicas eram complementares, de modo que uma técnica evidenciava parte dos falsos negativos da outra. Porém, eles não mencionam o ambiente ou a ferramenta utilizada para agregar esses resultados, induzindo o leitor que essa atividade foi realizada manualmente.

A visualização de software contribui para que as informações estruturais do software sejam abstraídas por meio de metáforas visuais, fornecendo uma ideia geral das estruturas, relacionamentos e lógica desse software (BENTRAD; MESLATI, 2011). Desse modo, o engenheiro de software pode utilizar de informações gráficas para identificar padrões que antes estavam escondidos no código fonte do sistema. Metáforas visuais utilizam símbolos, figuras geométricas, grafos, árvores e cores, entre outros recursos, para compor uma visão, capaz de representar as informações estruturais do sistema. Como exemplos de padrões que podem estar escondidos no código fonte têm-se as anomalias de modularização, ou *code smells* (FOWLER *et al.*, 1999)

- ***God Class:*** classes que violam os princípios de separação de interesses, por implementarem vários requisitos (RIEL, 1996);
- ***Lazy Class:*** classes que implementam pouca funcionalidade (FOWLER *et al.*, 1999);
- ***Divergent Change:*** classes que apresentam entrelaçamento de interesses e mudam de comportamento de acordo com o fluxo do sistema (FOWLER *et al.*, 1999).

Para a visualização de software há ferramentas, como por exemplo, MosaiCode (MALETIC *et al.*, 2011), PEVAD (ASPECTOS, 2007), $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013), VizzAspectJ-3D (BENTRAD; MESLATI, 2011) e SourceMiner (SILVA A. N.; CARNEIRO, 2012; CARNEIRO *et al.*, 2010).

Algumas ferramentas de visualização de software, como $SoftVis_{4CA}$, VizzAspectJ-3D e SourceMiner, evidenciam a importância de visualização das informações dos ITs de um software. As duas primeiras ferramentas realizam por si mesmas as atividades de descoberta das informações dos ITs, a fim de recuperar os dados a serem exibidos. Já a SourceMiner apresenta integração com a ferramenta de MIT chamada ConcernMapper (CARNEIRO *et al.*, 2010), de forma que os indícios de ITs identificados por essa ferramenta possam ser visualizados.

Para dar mais apoio ao engenheiro de software nas atividades de identificação e visualização de ITs no código fonte de um software, é interessante possibilitar a visualização dos resultados obtidos por diferentes ferramentas de MIT, em diversas visões fornecidas por diferentes ferramentas de visualização de software.

O engenheiro de software pode, com maior rapidez, dar manutenção ou evoluir um software se houvesse um ambiente capaz de possibilitar a comparação dos indícios de ITs encontrados e visualizados por meio de diferentes ferramentas. Assim, os pontos positivos de cada ferramenta de MIT poderiam ser combinados, além de possibilitar que novos padrões visuais sejam identificados por meio da comparação das informações exibidas.

1.2 Motivação e Objetivos

A integração de ferramentas de modo a permitir que vários índicos de ITs, encontrados por diferentes ferramentas de MIT, sejam visualizados por meio das metáforas em várias ferramentas de visualização, não é um processo trivial. Isso porque as ferramentas de MIT, normalmente, fornecem como interface de integração um arquivo de saída com os indícios de ITs identificados de acordo com um padrão específico da ferramenta, que muitas vezes é complexo, com particularidades e documentação escassa.

Por outro lado, nem sempre as ferramentas de visualização de software fornecem uma interface que permita a integração com as ferramentas de MIT. Em alguns casos, essa integração só é possível por meio de alterações no código fonte.

Do ponto de vista de uma análise realizada para encontrar os indícios de ITs, a comparação entre os resultados apresentados por diversas ferramentas de MIT proporciona que os pontos positivos de cada uma delas sejam combinados, propiciando que o resultado seja mais completo e consistente (CECCATO *et al.*, 2006). Uma forma de realizar essa comparação é por meio das diversas visões apresentadas pelas ferramentas de visualização de software. Essas visões, além de facilitar a comparação, ainda exibem informações complementares, de modo que o engenheiro de software poderia analisar como um IT afeta as diversas estruturas de um software.

Para possibilitar a integração de ferramentas de MIT com ferramentas de visualização de software, este projeto tem os seguintes objetivos:

1. Criar de um metamodelo para possibilitar a comunicação entre as ferramentas de MIT e de visualização de software. Desse modo, diversas ferramentas podem ser integradas de modo padronizado;
2. Facilitar a integração de ferramentas de MIT e de visualização de software;
3. Criar infra-estrutura capaz de facilitar a integração de diversas ferramentas de MIT e visualização de software. O esquema de um ambiente que utiliza-se dessa infra-estrutura pode ser visto na Figura 1.1;
4. Facilitar a comparação dos resultados obtidos por diferentes ferramentas de MIT por meio das visões de diferentes ferramentas de visualização de software;
5. Aumentar a precisão e a cobertura na identificação de indícios de ITs ao comparar os resultados de diversas ferramentas de MIT;
6. Apoiar a compreensão de software através da representação dos ITs que afetam as entidades de um software.

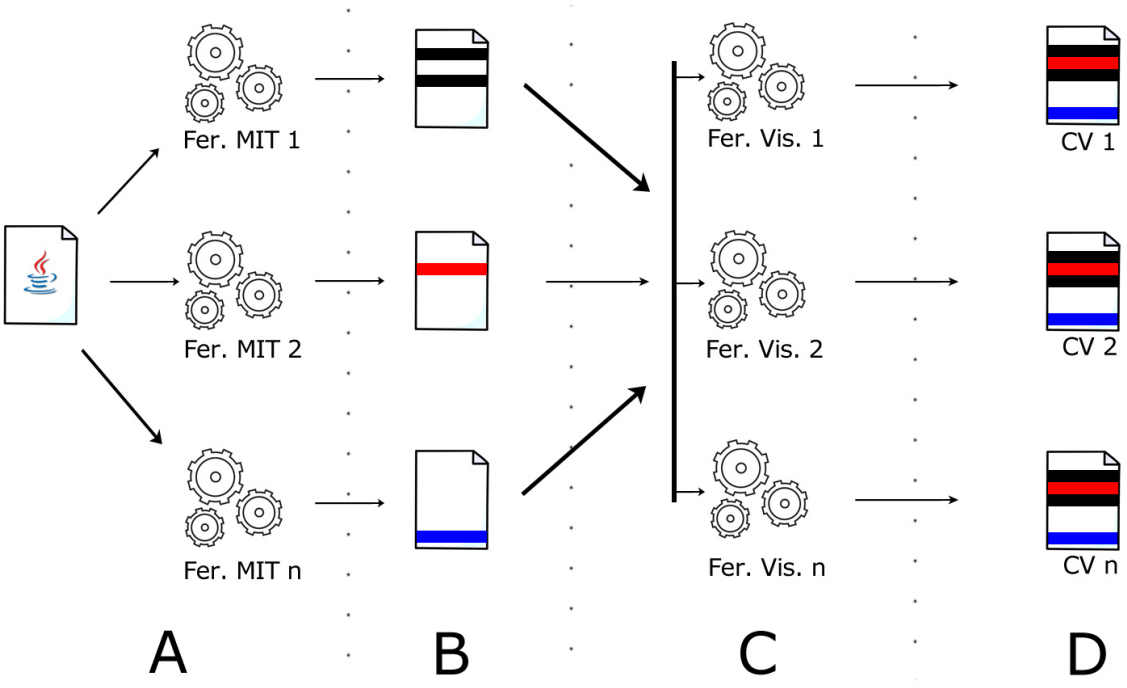


Figura 1.1: Esquema de uma arquitetura integrada de ferramentas de MIT e de visualização de software.

As letras colocadas na Figura 1.1 são usadas para explicar a sequência de passos que deve ser realizada durante a utilização do ambiente integrado:

A. O código fonte Java é analisado por diversas ferramentas de MIT;

B. Os indícios identificados por cada ferramenta de MIT são apresentados separadamente para o engenheiro de software. Alguns desses indícios podem ser identificados por mais de uma ferramenta;

C. Os resultados são analisados e agrupados por diversas ferramentas de visualização de software;

D. Os resultados são exibidos por meio dos diversos conjuntos de visões (CV) disponibilizados pelas ferramentas de visualização.

1.3 Metodologia de Desenvolvimento do Trabalho

Este trabalho utilizou ferramentas e técnicas para: i) mineração de interesses transversais e; ii) visualização de software. Diversas ferramentas foram analisadas, sendo que seis foram selecionadas para serem utilizadas, são elas: i) ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005); ii) ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a); iii) FINT (MARIN *et al.*, 2007); iv) XScan (NGUYEN *et al.*, 2011); v) $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013) e; vi) SourceMiner (SILVA A. N.; CARNEIRO, 2012; CARNEIRO *et al.*, 2010). As quatro primeiras são ferramentas de MIT e as duas últimas são ferramentas de visualização de software. Essas ferramentas foram selecionadas de acordo com o conhecimento deste autor e por algumas terem seus códigos fonte disponíveis.

As ferramentas de MIT selecionadas foram analisadas de forma a identificar as informações comuns utilizadas para representar os indícios de ITs. Já as ferramentas de visualização foram analisadas para que fosse identificado o conjunto mínimo de informações necessárias para que esses indícios fossem representados visualmente. De posse dessas informações foram analisados alguns projetos de pesquisa que definem metamodelos, visando a identificar características de um metamodelo efetivo para a representação visual de indícios de ITs.

Durante a revisão bibliográfica realizada não foi encontrado um metamodelo que pudesse ser utilizado neste projeto. Assim, com base no conhecimento adquirido, foi criado o metamodelo chamado *Indications Metamodel* (IMM) (TANNER *et al.*, 2013), para servir como um meio de comunicação entre as de MIT e de visualização de software. Esse metamodelo foi

utilizado para integrar as seis ferramentas selecionadas neste trabalho, para avaliar sua generalidade. Ao final desse processo foi criado um ambiente integrado, capaz de exibir os indícios de ITs encontrados pelas ferramentas de MIT por meio das visões das ferramentas de visualização. Provas de conceito foram realizadas para analisar esse ambiente e os resultados decorrentes de sua utilização.

Foram propostos dois experimentos para avaliar o IMM, planejados de acordo com os passos definidos por Wohlin *et al.* (2000) para: i) determinar se a utilização do IMM facilita a integração de ferramentas de MIT e de visualização de software e; ii) se a utilização em conjunto dessas ferramentas facilita o processo de identificação dos ITs, diminuindo o tempo total da análise e promovendo a identificação de ITs com melhor precisão e cobertura.

1.4 Trabalhos relacionados

Não foram encontrados na literatura específica trabalhos que propusessem uma forma de comunicação padrão entre as ferramentas de MIT e de visualização de software. Ceccato *et al.* (2006) evidenciam os benefícios da utilização em conjunto de diversas ferramentas de MIT. Os autores analisaram conjuntamente três ferramentas e concluíram que a identificação de falsos negativos é facilitada, visto que as técnicas eram complementares.

Alguns trabalhos, sobre ferramentas de visualização de software, evidenciam a necessidade de exibição das informações referentes aos ITs por meio das visões fornecidas por essas ferramentas. Algumas dessas ferramentas não apresentam uma forma de integração, como é o caso da $SoftVis_{4CA}$, da VizzAspectJ-3D e da SourceMiner. A SourceMiner proporciona duas formas de integração com as ferramentas de MIT, uma por meio do arquivo gerado pela ferramenta ConcernMapper e outro por meio de estrutura proposta por Silva A. N.; Carneiro (2012). No primeiro caso é necessário entender o formato do arquivo da ConcernMapper e implementá-lo. Esse formato não é trivial e apresenta pouca documentação (ROBILLARD, 2013; ROBILLARD; WEIGAND-WARR, 2005). No segundo caso é necessário implementar a integração para cada uma das ferramentas de MIT desejadas, sendo que essa solução é dependente da SourceMiner.

Piefel (2006) propõe um metamodelo para a geração de código fonte Java ou C++. No escopo deste trabalho, esse metamodelo poderia ser utilizado para a identificação dos trechos do código fonte que apresentam relação com um IT. Desse modo, poderiam ser identificadas as estruturas afetadas, mas nenhuma informação quanto ao IT seriam identificadas.

Bernardi e Lucca (2011) propõem um metamodelo para representar como um IT afeta as

estruturas estáticas de um software OO. Porém, esse metamodelo apresenta muitas informações não relevantes para este projeto, como o relacionamento entre as classes, métodos e heranças. Essas informações não são relevantes já que as ferramentas de MIT e de visualização de software que necessitam delas já possuem as rotinas necessárias para identificá-las.

1.5 Organização do Trabalho

Nos demais capítulos desta dissertação são apresentados os conceitos sobre a MIT e a visualização de software, a criação e utilização do metamodelo proposto, os experimentos realizados e os resultados obtidos.

- **Capítulo 2:** são apresentados os conceitos da Orientação a Aspectos e da MIT. Algumas das técnicas e ferramentas de MIT presentes na literatura são detalhadas, sendo que ao final desse capítulo é apresentada uma comparação entre essas técnicas.
- **Capítulo 3:** são abordados os conceitos da visualização de software, sendo apresentadas algumas das ferramentas de visualização presentes na literatura.
- **Capítulo 4:** é tratada a criação do metamodelo IMM, sendo apresentadas as informações utilizadas em sua concepção e as classes e funcionalidades presentes em sua versão programada.
- **Capítulo 5:** é abordada uma utilização do metamodelo apresentado no Capítulo 4, com a criação de um ambiente que integra as ferramentas ConcernMapper, ComSCId, FINT, XScan, $SoftVis_{4CA}$ e SourceMiner. De forma a avaliar esse ambiente é apresentado uma prova de conceito.
- **Capítulo 6:** são apresentados os detalhes e resultados de dois experimentos realizados para avaliar: i) o IMM como forma de integração entre ferramentas de MIT e de visualização de software e; ii) se a utilização do ambiente integrado auxilia na obtenção de um melhor resultado na procura por indícios de ITs.
- **Capítulo 7:** são comentadas as conclusões deste trabalho, apresentando as suas contribuições, limitações e alguns trabalhos futuros que podem ser realizados a partir deste.

Capítulo 2

ORIENTAÇÃO A ASPECTOS E MINERAÇÃO DE INTERESSES TRANSVERSAIS

2.1 Considerações Iniciais

O desenvolvimento com Orientação a Objetos (OO) surgiu com o propósito de melhorar a modularização dos sistemas computacionais, proporcionando mais enfoque nos dados do que nos processos. Essa melhor modularização possibilitou um aumento no reúso de componentes de software, proporcionando maior agilidade no processo de desenvolvimento e reduzindo os custos com a manutenção. Assim mesmo, ainda existem alguns requisitos do software que não são completamente encapsulados de acordo com as regras do paradigma OO (KICZALES *et al.*, 2001).

Na maioria dos softwares desenvolvidos ocorre o espalhamento e entrelaçamento dos requisitos implementados, de acordo com as necessidades de seus `stakeholders`. Um requisito está espalhado pelo código fonte quando as chamadas das funções que o implementam estão presentes em vários módulos do software. Já um requisito entrelaçado é representado pela chamada das funções da classe que o implementa em uma outra classe do sistema. Esses requisitos, espalhados e entrelaçados, entrecortam transversalmente os demais módulos do sistema e são chamados de Interesses Transversais (IT).

Para amenizar o problema do espalhamento e o entrelaçamento dos ITs, surge a Programação Orientada a Aspectos (POA) (KICZALES *et al.*, 1997), como um complemento a OO, possibilitando que os ITs sejam encapsulados em unidades lógicas chamadas de aspectos. Essas novas unidades proporcionam melhor modularização como também melhor manutenibilidade e facilidade no processo de reúso (KICZALES *et al.*, 1997) dos componentes de um software.

A utilização de OA não é um processo trivial, sendo que o código fonte de um sistema deve ser cuidadosamente analisado, de modo que os ITs sejam identificados (ROBILLARD; WEIGAND-WARR, 2005). Geralmente os sistemas possuem milhares de linhas de código e nem sempre apresentam uma documentação atualizada, tornando a análise do código fonte inviável. Para auxiliar esse processo surgem técnicas de mineração que guiam o engenheiro de software durante o processo de descoberta dos ITs (ZHANG; JACOBSEN, 2012).

Este capítulo apresenta conceitos relacionados a orientação a aspectos e mineração de interesses transversais utilizados neste trabalho, e sua organização é descrita a seguir. Na Seção 2.2 são apresentados os conceitos principais da Programação Orientação a Aspectos. Na Seção 2.3, as principais características da Mineração de Interesses Transversais são descritas, citando técnicas e ferramentas encontradas na literatura. Por fim, na Seção 2.4, é apresentado um comparativo entre todas as técnicas de mineração de ITs apresentadas e são discutidas as considerações finais deste capítulo.

2.2 Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) (KICZALES *et al.*, 1997) tem como objetivo complementar a decomposição proporcionada pela OO fornecendo uma forma de encapsular os ITs em unidades lógicas chamadas de aspectos. Dessa forma, POA permite que os requisitos funcionais do sistema sejam implementados utilizando-se uma linguagem de programação OO, enquanto os ITs são implementados por uma linguagem OA. Com isso, os componentes OO de um sistema terão suas funcionalidades complementadas pelas funcionalidades implementadas pelos aspectos. A atividade de complementar ou combinar as funcionalidades dos módulos de um sistema com as dos aspectos é responsabilidade do combinador ou Weaver (BÖLLERT, 1999). Esse processo é capaz de, por meio de regras definidas dentro dos aspectos, identificar quais módulos OO devem ser atualizados e realizar as alterações necessárias adequadamente.

Várias linguagens para POA existem com diferentes enfoques. Para a linguagem Java, pode-se citar: AspectJ (KICZALES *et al.*, 2001) e CeaserJ (ARACIC *et al.*, 2006). Para a linguagem C tem-se a linguagem AspectC (COADY *et al.*, 2001) e para a linguagem C++ a AspectC++ (SPINCZYK *et al.*, 2002). Cada uma dessas linguagens fornece as estruturas necessárias para que os desenvolvedores possam se utilizar dos benefícios proporcionados pelo paradigma AO, que são:

- **Ponto de Junção:** ponto bem definido na execução do sistema que pode ser entrecortado para a execução das funcionalidades do aspecto;

- **Ponto de corte:** definição de quais pontos de junção devem ser entrecortados; Adendo: funcionalidade do aspecto a ser executado sempre que um ponto de corte for alcançado na execução do sistema;
- **Aspecto:** estrutura semelhante a uma classe que agrupa os pontos de corte e os adendos, encapsulando as funcionalidades de um IT.
- **Weaver:** processo responsável por combinar as funcionalidades implementadas pelos aspectos com as funcionalidades do sistema, de acordo com as definições dos pontos de corte.

2.3 Mineração de Interesses Transversais

A Mineração de Interesses Transversais (MIT) visa à identificação de trechos do sistema pertencentes a requisitos que estão espalhados e entrelaçados no código fonte (ZHANG; JACOBSEN, 2012). Essa atividade segue os padrões definidos por uma ou mais técnicas de mineração, e normalmente é auxiliada por uma ferramenta que implementa essa(s) técnica(s).

Na Seção 2.3.1 são apresentadas algumas técnicas de MIT que constam na literatura, enquanto que na Seção 2.3.2 são ilustradas as ferramentas que implementam essas técnicas.

2.3.1 Técnicas de Mineração de Interesses Transversais

Uma técnica de MIT define os passos que o engenheiro de software deve seguir para identificar os ITs que afetam um sistema. Independente da técnica utilizada, sempre é necessária a intervenção do engenheiro de software para realizar algum passo ou analisar os resultados apresentados. Algumas das técnicas de mineração presentes na literatura específica são apresentadas com as suas principais características e um exemplo de sua utilização.

2.3.1.1 Análise Exploratória

A técnica de análise exploratória (ROBILLARD; MURPHY, 2002) cuida da identificação manual dos indícios de ITs, ficando sob a responsabilidade do engenheiro de software explorar todo o código fonte a fim de encontrar os indícios de ITs. Essa técnica define que um conjunto inicial de indícios (chamado de conjunto de sementes) seja previamente selecionado com a finalidade de iniciar as atividades de busca no código fonte do sistema. Cabe ao engenheiro de software analisar esse conjunto de sementes e procurar por características similares entre elas

no código fonte. Após encontrar essas características, uma busca detalhada no sistema deve ser realizada para verificar a existência de outras estruturas que possam também apresentar essas características e não foram encontradas anteriormente.

Uma semente pode ser um método ou um atributo de uma classe. Se for um método, a busca deve ser pela sua declaração e por suas chamadas. Se forem atributos, a busca é por sua declaração e atividades de escrita e leitura que os utilizem. Essa técnica exige suficiente conhecimento prévio do sistema por parte do engenheiro de software, de modo que seja capaz de compreender todo o sistema, a fim de selecionar um adequado conjunto inicial de sementes e analisá-lo para identificar suas características comuns.

Para auxiliar na detecção de sementes Robillard e Murphy (2002) propõem a utilização de grafos, para abstrair as principais informações estruturais da implementação de um interesse. Esses grafos foram chamados de Grafos de Interesses (*Concern Graphs*) e são baseados em um modelo do sistema, criado a partir das relações de uma semente com as demais estruturas do sistema.

A construção de um Grafo de Interesses ocorre da seguinte forma: 1) uma semente é selecionada e representada como vértice de um grafo; 2) são analisados os pontos do código fonte em que essa semente é declarada e utilizada, representados como outros vértices do grafo; 3) as setas direcionadas são então adicionadas para representar a relação entre esses nós, de modo que a seta liga o nó no qual a semente é declarada ao nó onde ela é utilizada, com direção para este segundo nó.

A ferramenta FEAT (ROBILLARD; MURPHY, 2007; ROBILLARD; MURPHY, 2003), apresentada na Seção 2.3.2.1, é uma ferramenta que auxilia na utilização da análise exploratória e utiliza Grafos de Interesses. A ferramenta ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005) é uma outra que auxilia na utilização da análise exploratória, porém não utiliza Grafos de Interesses, e é apresentada na Seção 2.3.2.2.

2.3.1.2 Análise de *Fan-in*

A técnica de análise de *Fan-in* (MARIN *et al.*, 2007) utiliza o conceito de que os métodos com alto valor de *Fan-in* são os que estão mais espalhados pelo código fonte. De posse desse valor, cabe ao engenheiro de software, com base em seu conhecimento sobre o sistema, analisar quais os métodos que apresentam indícios de ITs. Para auxiliar nessa análise, o engenheiro de software pode determinar qual o valor mínimo de *Fan-in* a ser utilizado como filtro dos métodos a serem exibidos como resultado.

Também é sua função analisar o conjunto de métodos retornados e identificar se realmente são indícios de um IT. Se sim, ainda é necessário que o engenheiro de software classifique qual IT esse método representa, visto que essa técnica não é capaz de apresentar seus resultados separados por IT.

A ferramenta FINT (MARIN *et al.*, 2007) implementa essa técnica e é apresentada na Seção 2.3.2.3.

2.3.1.3 Análise por Detecção de Clones

Técnicas para detecção de clones (BRUNTINK *et al.*, 2005), têm por base a premissa que, para implementar um interesse transversal, sem a utilização de OA, é necessário "copiar e colar" um determinado conjunto de linhas e espalhá-las pelo código fonte do sistema. Além de que, em alguns casos, somente pequenas alterações são necessárias. Algumas das técnicas de detecção de clones encontradas na literatura, (BRUNTINK *et al.*, 2005) são:

- **Baseadas em texto** (DUCASSE *et al.*, 1999): procuram por trechos de código idênticos ou similares, realizando pouca ou nenhuma transformação no código fonte antes de iniciar as buscas. Normalmente ignoram espaços brancos e comentários;
- **Baseadas em tokens** (KAMIYA *et al.*, 2002): realizam a busca por meio de uma análise léxica, sendo que o código fonte deve ser transformado em *tokens*, antes das buscas serem iniciadas.
- **Baseadas em AST** (BAXTER *et al.*, 1998): utilizam de representações sintáticas do código fonte, normalmente uma *Abstract Syntax Tree* (AST), para procurar por sub-árvores similares.
- **Baseadas em PDG** (KRINKE, 2001): realizam a busca com o auxilio de um *Program depence graph* (PDG), que é uma abstração do código fonte que contém informações semânticas do sistema, como controle e fluxo de execução, sendo os clones identificados por meio de sub-grafos iguais ou parecidos.
- **Baseadas em métricas** (MAYRAND *et al.*, 1996): dividem o código fonte em trechos e, para cada trecho, é atribuída uma pontuação de acordo com algumas métricas. Essa pontuação é armazenada em uma *hash*, que é então utilizada para encontrar os clones.

CCFinder (KAMIYA *et al.*, 2002) implementa a técnica de detecção de clones baseada em *tokens* e é apresentada na Seção 2.3.2.4.

2.3.1.4 *Concern Peers*

A técnica de análise por *Concern Peers* (NGUYEN *et al.*, 2011) tem por base que partes similares do sistema possuem propriedades similares, sendo que os métodos que possuem interações similares têm uma tendência de pertencerem a um mesmo IT. Essa técnica define que dois métodos, que possuem similaridade suficiente, são chamados de *Concern Peers*, ou como tratado no contexto deste trabalho, um indício de IT. De acordo com essa técnica, os pares de *Concern Peers* de um software, após sua detecção, são agrupados de acordo com suas similaridades e ordenados de acordo com o número total de interações relevantes de seus membros. Logo após, os grupos com maior número de interações relevantes, são recomendados como possíveis ITs presentes no software.

Os autores desta técnica recomendam:

- **Mineração de aspectos:** essa técnica, em softwares não orientados a aspectos, é responsável por indicar os grupos que possuem maiores números de interações como possíveis ITs presentes no software;
- **Atualização de aspectos:** softwares que já foram refatorados utilizando a OA ou softwares que inicialmente foram desenvolvidos com OA, ao passarem por um processo de manutenção devem ser novamente analisados. Essa análise verifica se nenhum novo IT foi adicionado ao software ou se a modularização de algum dos ITs já presentes no software foi quebrada.

A ferramenta XScan (NGUYEN *et al.*, 2011) implementa essa técnica e é apresentada na Seção 2.3.2.5.

2.3.1.5 Tipos e Texto

A técnica de análise por tipos e texto (HANNEMANN; KICZALES, 2001) visa à identificação dos indícios de interesses transversais por meio de sua classe (tipo) ou por um conjunto de caracteres (texto ou *string*). Essa técnica determina que um conjunto inicial de regras seja cadastrado, de forma que possam ser utilizadas para a realização das buscas.

As regras de tipo determinam as classes, cujas declarações de seus objetos devem ser identificadas como indícios de IT. Por exemplo, ao se criar a regra de que as classes `java.sql.Connection` e `java.sql.ResultSet` pertencem ao IT de persistência em banco de dados, tem-se que toda classe ou método que instanciar um objeto dessas classes estará implementando esse IT, e portando são um indício.

Já as regras de texto são utilizadas para indicar conjuntos de caracteres que indicam a presença de um IT em nomes de classes, interfaces, métodos, atributos e valores atribuídos as variáveis do tipo `String`. PARREIRA JÚNIOR (2011) comenta que essa técnica é conveniente para sistemas que apresentam grande padronização de nome de identificadores. Como exemplo, pode-se criar uma regra para indicar que qualquer classe, interface, método e variável cujo nome contenha a cadeia de caracteres "`sql`" está relacionada ao IT de persistência em banco de dados. Como outro exemplo, pode-se relacionar a cadeia de caracteres "`INSERT INTO`" com esse mesmo IT. Com isso, qualquer variável do tipo `String` que receber como valor uma cadeia de caracteres que contenha "`INSERT INTO`" como parte deste valor, estará implementando o IT de persistência em banco de dados.

Essa técnica exige conhecimento prévio do sistema e do IT analisado, de modo que o do conjunto de regras possa ser criado. Se as regras criadas não forem adequadas, por conta da automação apresentada por essa técnica, podem ocorrer muitos falsos positivos e falsos negativos. Falsos positivos são trechos de código fonte que foram identificados mas que não apresentam relação com um IT. Já os falsos negativos são trechos de código fonte que deveriam ser identificados mas que não foram encontrados durante as buscas realizadas.

Como exemplo de falso positivo, considere o seguinte cenário. As operações realizadas em banco de dados necessitam de comandos em linguagem SQL armazenados em variáveis do tipo `String`, portanto é criada a regra de que qualquer variável do tipo `String` é um indício do IT de persistência. Como a classe `String` é uma das mais utilizadas, vários falsos positivos seriam identificados. Como exemplo de falsos negativos considere o seguinte cenário: a regra utilizada para identificar o IT de *logging* determina que classes que possuam em seu nome o conjunto de caracteres "`log`" sejam identificadas. Porém, em um sistema, como convenção, foi utilizado o conjunto de caracteres "`registro`" para identificar as funções que implementam esse IT. Logo, vários trechos de código fonte deixariam de ser identificados.

A ferramenta ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a) implementa essa técnica e é apresentada na Seção 2.3.2.6.

2.3.2 Ferramentas de Mineração de Interesses Transversais

Ferramentas de MIT são apoios computacionais que implementam uma ou mais técnicas de mineração, com o objetivo de auxiliar o engenheiro de software em sua utilização durante o processo de identificação dos indícios de ITs. Esta seção descreve algumas ferramentas que implementam as técnicas de mineração descritas na seção anterior.

2.3.2.1 FEAT

FEAT (*Feature Exploration and Analysis Tool*) (ROBILLARD; MURPHY, 2007; ROBILLARD; MURPHY, 2003) auxilia na utilização da técnica de análise exploratória com o auxílio dos Grafos de Interesses. Foi desenvolvida como um *plug-in* do Eclipse (Eclipse Foundation, 2013) e é responsável por auxiliar o engenheiro de software na criação, explorção e atualização dos Grafos de Interesses. Um tela dessa ferramenta é apresentada na Figura 2.1. Na área (A) pode-se ver a declaração do Grafo de Interesse, para a identificação do interesse transversal de persistência. As sementes adicionadas ao grafo são apresentados na área (B), tais como as sementes representadas pela classe `GerenteConexao`, seu atributo `connection` e seus métodos `getConecta()` e `stopConnection()`. Ao selecionar uma semente, no caso a função `getConecta()`, pode-se visualizar na área (C) as informações de relacionamento dessa semente com as demais estruturas do sistema. Uma informação de relacionamento pode ser: i) onde a semente é declarada; ii) quais demais estruturas do sistema ela chama, acessa, sobrescreve ou utiliza; iii) quais estruturas do sistema sobrescrevem a semente e; iv) quais estruturas do sistema possuem uma chamada a essa semente. Cabe, então, ao engenheiro de software analisar todas essas informações, identificando novas sementes e analisando as novas informações de relacionamento, até que todo o sistema seja analisado.

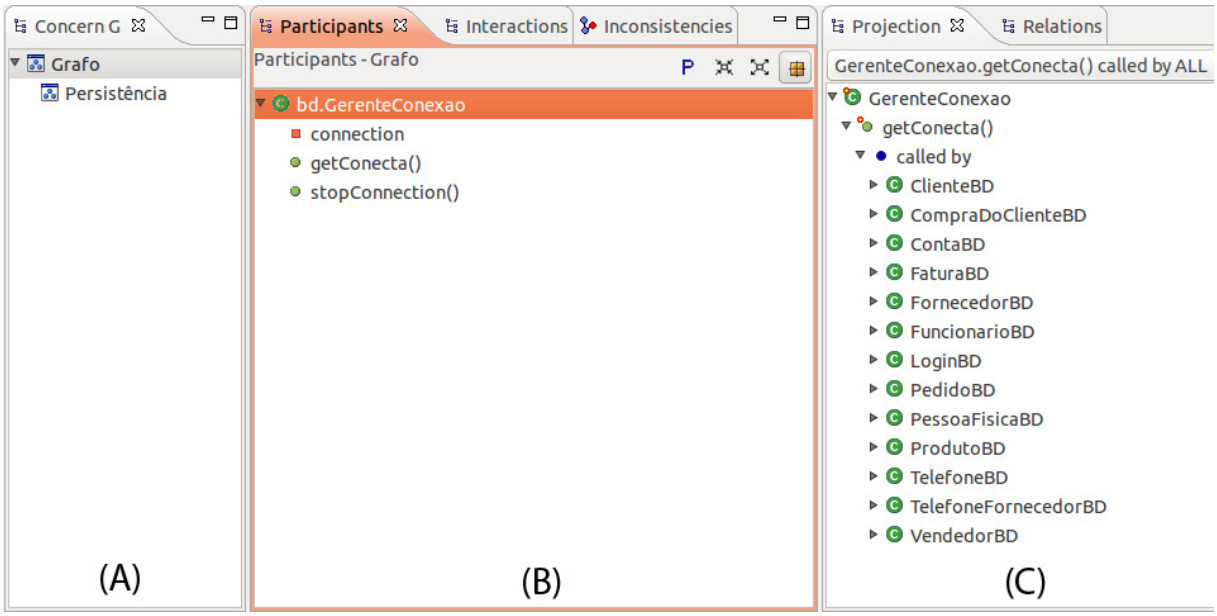


Figura 2.1: Tela da ferramenta FEAT.

2.3.2.2 ConcernMapper

A ferramenta ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005) foi desenvolvida como um *plug-in* do Eclipse e tem como um de seus desenvolvedores Robillard, que também desenvolveu a FEAT. Ele afirma que a ConcernMapper foi desenvolvida a partir dos conhecimentos adquiridos no desenvolvimento e utilização da FEAT.

A ConcernMapper auxilia na utilização da técnica de mineração por análise exploratória, mantendo uma listagem de todas as sementes encontradas e facilitando as buscas no código fonte. As buscas podem ser: i) pelo ponto no qual a semente é declarada; ii) pelos pontos nos quais a semente é chamada, no caso de um método e; iii) pelos pontos nos quais ocorrem atividades de leitura e escrita, no caso de um atributo. Essas buscas podem ter como escopo: i) todos os projetos do *workspace* do Eclipse; ii) somente o mesmo projeto da semente e; iii) a hirarquia da classe da semente.

Um exemplo de sua utilização pode ser visto na Figura 2.2. Neste exemplo, é analisada a classe `GerenteConexao` (área A), responsável por cuidar das conexões com o banco de dados. Essa classe possui a função `getConecta()`, responsável por retornar uma conexão aberta para o banco de dados. Esssa função foi selecionada como semente, e portanto, aparece listada na área B, responsável por exibir todas as sementes identificadadas, separadas por IT. É dever do engenheiro de software separar as sementes por IT no momento em que as for adicionando. A área C tem por finalidade apresentar o resultados das buscas, que no caso desse exemplo, apresenta todas as classes do sistema que utilizam a função `getConecta()`. A partir desse momento cabe ao engenheiro de software analisar todas essas classes, adicionando novas sementes e realizando novas buscas, até analisar o sistema todo.

2.3.2.3 FINT

A técnica de mineração por *Fan-in* é implementada pela ferramenta FINT (MARIN *et al.*, 2007), a qual é responsável por realizar a contagem do valor *Fan-in* de todos os métodos do sistema, apresentar os resultados organizados por nome do método ou por valor de *Fan-in*.

Os resultados, comentados anteriormente devem ser filtrados por: i) qualquer método do sistema para que o cálculo de seus valores *Fan-in* sejam ignorados; ii) qualquer método para que eles não sejam levados em consideração na hora de realizar o cálculo do *Fan-in* dos demais métodos do sistema; iii) qualquer método que tenha valor *Fan-in* de um valor mínimo, ou *Threshold*; iv) todos os métodos acessores de um sistema e; v) métodos implementados por bibliotecas. A interface da tela de filtros pode ser vista na Figura 2.3.

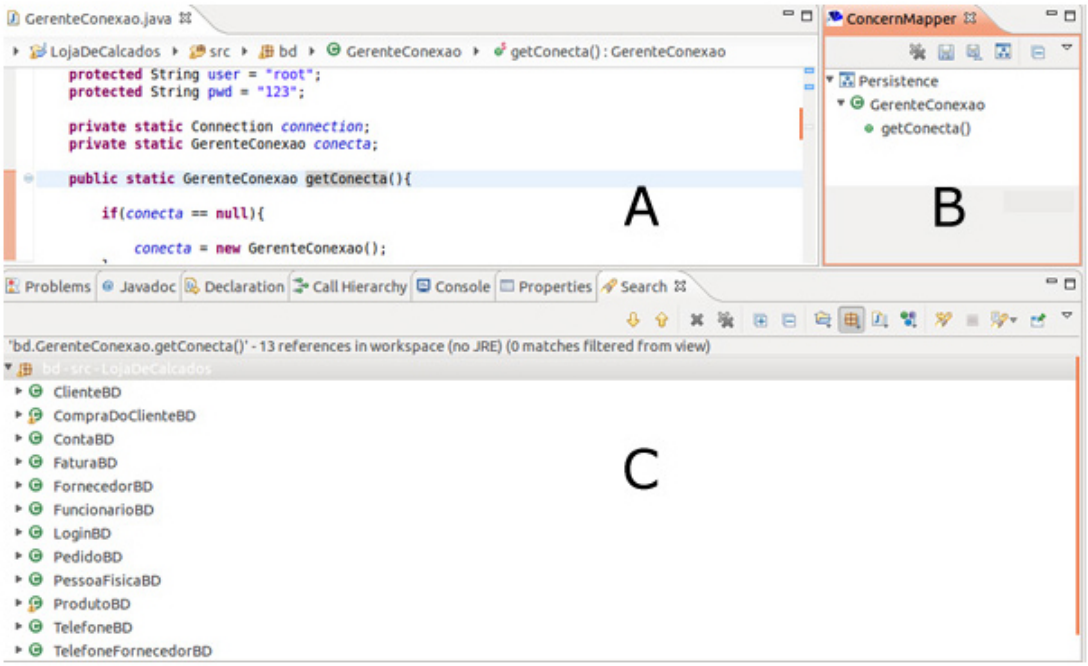


Figura 2.2: Exemplo de utilização da ConcernMapper.

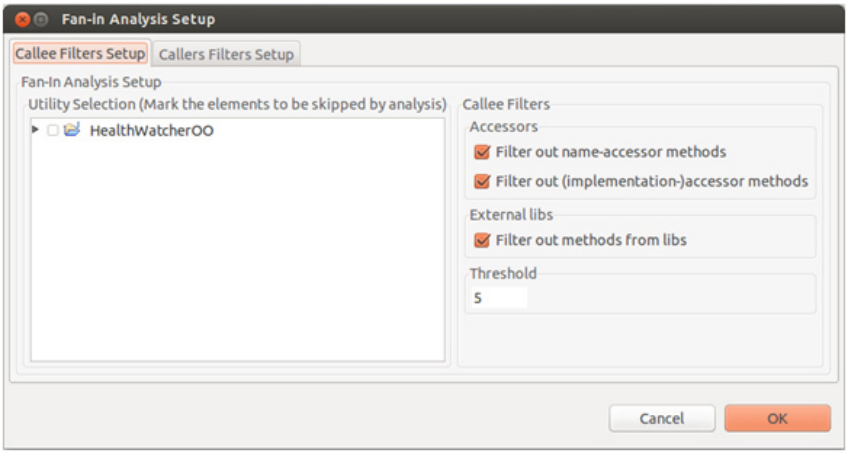


Figura 2.3: Interface de filtros da ferramenta FINT.

Os resultados obtidos após o cálculo do *Fan-in* são visualizados na aba *Fan-in Analisys View* (Figura 2.4). Neste exemplo o *Threshold* foi definido como cinco e os métodos estão ordenados por ordem alfabética. Essa visão exibe os resultados em forma de árvore, sendo que os métodos chamadores são agrupados pelo método chamado.

2.3.2.4 CCFinder

A ferramenta CCFinder (KAMIYA *et al.*, 2002) foi implementada na linguagem de programação C++ e pode identificar clones em sistemas implementados nas linguagens C, C++, Java e COBOL. Ela implementa a técnica de detecção de clones baseada em *tokens*. Após dividir o

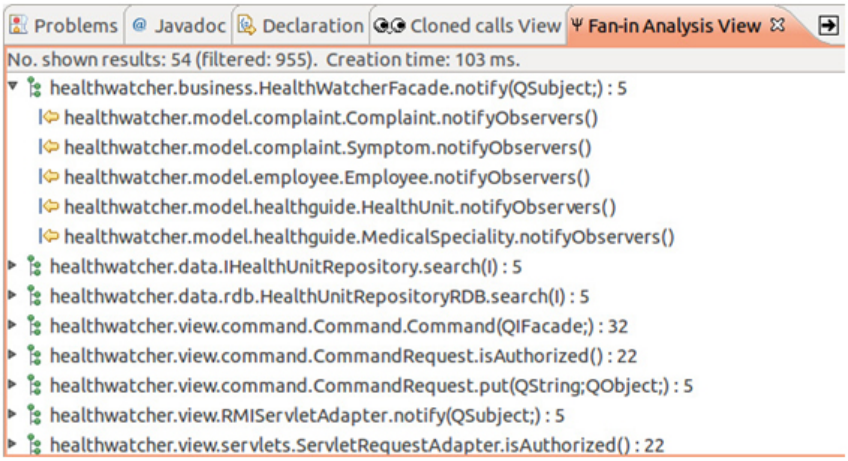


Figura 2.4: Exemplo de resultados exibidos pela ferramenta FINT.

código fonte em *tokens* (considerando as regras léxicas da linguagem de programação), realiza duas transformações nesse conjunto: i) transformação da sequência dos *tokens*, adicionando, removendo ou alterando *tokens* de acordo com as regras definidas por Kamiya *et al.* (2002) e; ii) os identificadores de parâmetros, constantes e variáveis são trocados por *tokens* especiais, possibilitando que códigos com diferentes nomes de variáveis sejam identificados como clones.

Kamiya *et al.* (2002) apresentam alguns casos de estudo com a ferramenta. Um deles descreve a análise de clones presentes no código do JDK (*Java Development Kit*), sendo encontrado um total de 2333 clones. Outro caso de estudo descreve a procura por clones e similaridades entre os códigos fonte dos sistemas operacionais Linux, FreeBSD e NetBSD. Como resultado, foi descoberto que cada sistema operacional possui grande quantidade de clones dentro de seus códigos fonte, que os sistemas operacionais FreeBSD e NetBSD possuem uma grande quantidade de clones comuns entre si, e que poucos desses clones são encontrados no Linux.

2.3.2.5 XScan

A ferramenta XScan (NGUYEN *et al.*, 2011) implementa a técnica de mineração por *Concern Peers* e tem a responsabilidade de analisar todo o código fonte, realizando operações complexas. Dentre elas: i) detecção de clones; ii) análise de *Fan-in*; iii) análise dos Grafos de Interesse e; iv) identificação dos *peers methods*, ou seja, métodos com características similares que irão compor os grupos de *Concern Peers*.

XScan foi desenvolvida como um *plug-in* do Eclipse, porém, para a sua execução, deve-se alterar em seu código fonte o *path* e o nome do projeto a ser analisado. É necessário realizar a sua recompilação sempre que um novo projeto for analisado. Após executar a ferramenta, os resultados são apresentados em uma pasta chamada *Output*, que é criada no mesmo *path* que o

do *workspace* do Eclipse.

Nesta pasta encontram-se dois arquivos: i) "AspectWiz - Peer Groups.txt" e; ii) "CBFA - CBFA Groups.txt". O primeiro arquivo apresenta os grupos formados por meio da combinação das técnicas apontadas no primeiro parágrafo. Já o segundo arquivo utiliza somente a técnica de agrupamento baseada no valor de Fan-in dos métodos analisados (*Cluster Based Fan-in Analysis* – CBFA). Os autores afirmam que o primeiro arquivo possui resultados melhores por utilizar mais técnicas para a identificação dos grupos.

Os arquivos de resultado apresentam os grupos ordenados por *rank*. O *rank* de um grupo indica a probabilidade dele pertencer a um IT, sendo que quanto maior o rank maior será essa probabilidade. Esses arquivos, por serem textuais, não apresentam uma forma de *link* para o código fonte. O engenheiro de software deve analisar os grupos apresentados e, se desejar visualizar o código fonte de um método, abrir manualmente a classe em que ele se encontra. Outra desvantagem é que caracteres específicos são misturados para separar as estruturas (pacote, classe e parâmetro) do caminho de um método. O engenheiro de software é que tem a responsabilidade de decifrá-los, para identificar qual método está sendo apresentado como resultado.

Essa técnica apresenta um alto grau de automação, sendo que não é aceita nenhuma intervenção do engenheiro de software, que somente pode analisar os resultados apresentados. Se houver a necessidade de refinamento desses resultados, o mesmo deve ser realizado manualmente ou com o auxílio de uma outra ferramenta.

2.3.2.6 ComSCId

ComSCId (*Computacional Support for Concern Identification*) (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a) foi desenvolvida como um *plug-in* do Eclipse e implementa a técnica baseada em tipos e texto. Essa ferramenta tem uma interface (Figura 2.5) para que o engenheiro de software possa incluir, alterar e remover as regras utilizadas nas buscas. Essa regras são armazenadas em um arquivo XML, permitindo que cada sistema possua seu conjunto particular de regras, ou que essas regras sejam compartilhadas entre diversos sistemas.

A utilização dessa ferramenta deve ser iterativa, de modo que um conjunto de regras seja cadastrado para que a busca posterior seja realizada a partir delas. O engenheiro de software deve analisar os resultados obtidos e decidir entre: i) alterar o conjunto de regras e executar uma nova iteração ou; ii) finalizar as buscas.

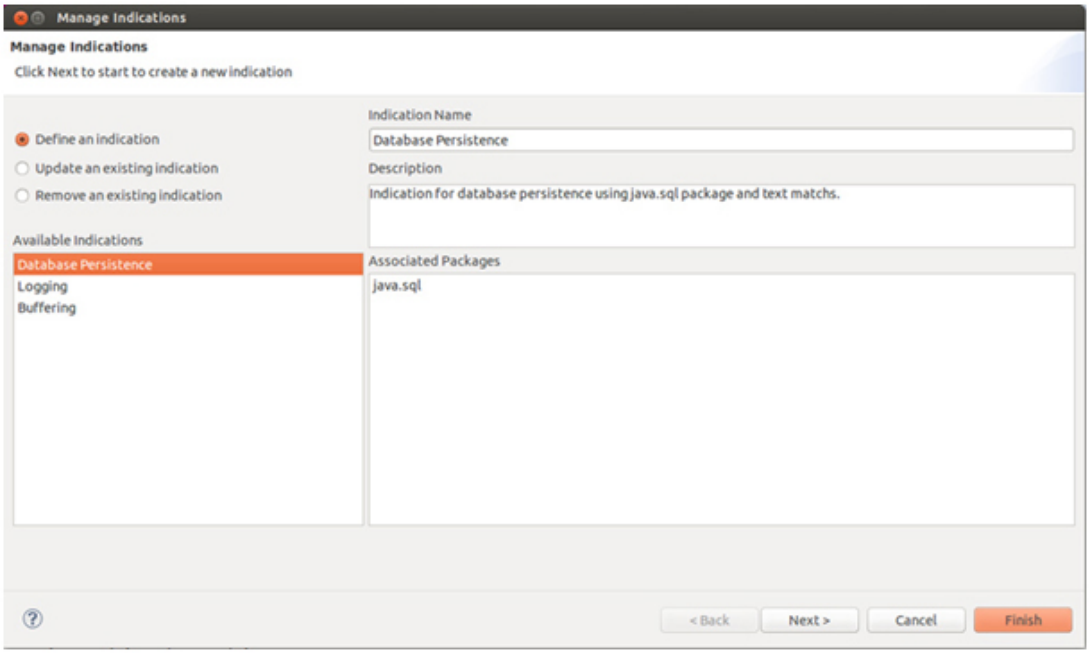


Figura 2.5: Interface para cadastrar, alterar e remover regras.

Para auxiliar na identificação dos ITs, a ComSCId fornece um conjunto pré-cadastrado de regras para três indícios: i) persistência em banco de dados (*database persistence*); ii) *logging* e; iii) *buffering*. Esses conjuntos visam a auxiliar a execução da primeira iteração e possuem a mesma deficiência que a técnica de mineração por tipos e texto, sendo que sistemas que não possuem uma forte padronização nos nomes de seus identificadores apresentarão poucos resultados significativos.

Para exibir os indícios encontrados, a ComSCId utiliza uma visão em forma de árvore (Figura 2.6), na qual os indícios são apresentados logo abaixo das classes onde foram encontrados. Cabe ao engenheiro de software expandir todos os pacotes e classes do sistema, procurando pelos indícios de ITs encontrados. Essa visão apresenta como problemas: i) somente exibe os indícios de um IT; ii) para que os métodos afetados sejam visualizados é necessário expandir o nó do pacote e da classe; iii) difícil visualização geral dos pacotes, classes e métodos afetados pelos indícios de ITs e; iv) difícil visualização de todas as classes do sistema que são afetadas por mais um IT.

Por ser um processo de buscas automático, podem ocorrer falsos positivos e falsos negativos, como comentado na Seção 2.3.1.5. Para tentar evitar isso, PARREIRA JÚNIOR *et al.* (2010a) propuseram que mais um passo fosse incluído à abordagem iterativa de buscas por indícios de ITs. Os resultados obtidos, além de serem analisados na visão de árvore, devem também ser visualizados em um diagrama de classes anotadas. Após a análise do diagrama é que o engenheiro de software deve decidir se deve atualizar o conjunto de regras ou finalizar a

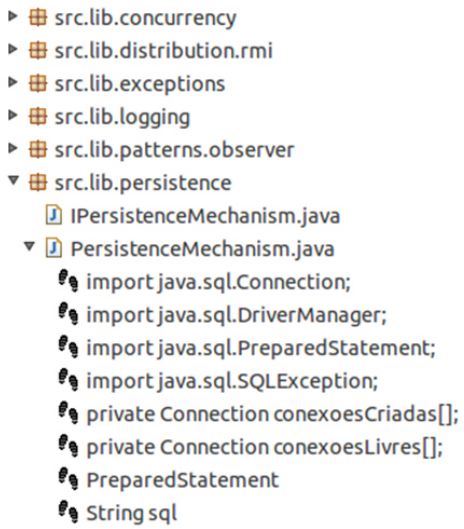


Figura 2.6: Apresentação dos indícios de persistência na visão de árvore.

busca.

2.4 Considerações Finais

Mesmo com a modularização proporcionada pela OO, ainda persistem problemas como o espalhamento e o entrelaçamento de algumas funcionalidades do sistema, os chamados ITs. Visando melhorar essa situação surge a POA, que tem como objetivo encapsular os ITs em unidades modulares chamadas de aspectos. Esse processo não é trivial e necessita de apoio, proporcionados pelas técnicas e ferramentas de MIT. Nas seções anteriores foram apresentadas as principais características de algumas técnicas e ferramentas de MIT presentes na literatura específica. Essas características são sumarizadas na Tabela 2.1, a fim de facilitar a comparação entre os pontos positivos e negativos de cada uma das técnicas apresentadas.

Como pode ser visto na Tabela 2.1, não existe uma técnica perfeita, sendo que todas possuem seus pontos positivos e negativos. Assim, seria interessante realizar uma abordagem que combinasse os pontos positivos de cada técnica e que possibilitasse a visualização dessa combinação por meio das visões disponibilizadas por ferramentas de visualização de software, que serão apresentadas no próximo capítulo.

Tabela 2.1: Sumarização dos pontos positivos e negativos das Técnicas de MIT.

	Análise Exploratória	**Análise por Fan-in**	**Análise por detecção de Clones**	**Concern Peers**	**Tipos e Texto**
Tempo e esforço	Alto	Médio	Baixo	Baixo	Baixo
Grau de automação	Baixo	Médio	Alto	Alto	Médio
Conhecimento necessário da estrutura do sistema	Alto	Médio	Baixo	Baixo	Médio
Resultados separados por IT	Sim	Não	Não	Não	Sim
Depende de regras de nomenclatura	Não	Não	Não	Não	Sim

Capítulo 3

VISUALIZAÇÃO DE SOFTWARE

3.1 Considerações Iniciais

Uma das dificuldades enfrentadas pelos engenheiros de software está na identificação e no entendimento das estruturas que compõem os softwares. Como o ser humano possui habilidade de absorver informações por meio da visão (WARE, 2004), foram desenvolvidos meios que fornecem uma visualização abstrata do software de modo que seja mais fácil compreendê-lo (BENTRAD; MESLATI, 2011).

A modelagem de software, que consiste na criação de modelos que representam visões da estrutura e do comportamento de um software (PRESSMAN, 2001), é uma das estratégias usadas para visualização de problemas/soluções. Esses modelos auxiliam na definição dos elementos que irão compor o software durante o seu desenvolvimento. Contudo, os modelos podem se tornar obsoletos após o software passar por processos de manutenção, uma vez que eles raramente são atualizados (DEMEYER *et al.*, 2005). Como esses modelos são artefatos estáticos e há pouca ou nenhuma interação com os engenheiros de software, recomenda-se o uso de técnicas que permitam visualizar e analisar a estrutura do software a partir do seu código-fonte.

Nesse contexto, a Visualização de Software (DIEHL, 2002) surge como uma técnica que possibilita a representação das informações contidas no software por meio de elementos visuais. Diferentes visões do conteúdo, da lógica, da estrutura e dos relacionamentos entre as entidades que compõem o software podem ser obtidas (BENTRAD; MESLATI, 2011). Desse modo, os elementos do software podem ser visualizados exatamente como estão implementados no código-fonte e informações podem ser mais facilmente identificadas com o apoio de ferramentas de visualização.

A visualização de software torna-se muito útil nos casos de manutenção de software, con-

siderando que a análise exaustiva do código fonte é ineficiente e impraticável, pelo esforço e prazos que essa atividade demanda. Na tentativa de amenizar essa situação, a visualização de software possibilita, por meio de suas metáforas visuais, que alguns padrões não perceptíveis no código fonte de um software sejam identificados pelos engenheiros de software, reduzindo o esforço e o tempo gastos na sua análise.

A utilização de visualização de software se torna bastante adequada para as atividades de manutenção perfectiva, auxiliando na identificação de anomalias de modularização, ou *code smells* (FOWLER *et al.*, 1999). *Code smells* não são *bugs*, uma vez que não afetam a execução de um software, mas ferem as regras de uma boa codificação e modularização, dificultando os processos de manutenção. Como exemplos de *code smells* podem ser citados: *God Class*, *Lazy Class*, *Featury Envy*, *Divergent Change* e *Too many parameters*.

Neste capítulo são apresentados os conceitos relacionados às ferramentas de visualização de software utilizadas neste trabalho, organizados da seguinte forma: na Seção 3.2 são apresentados os conceitos da visualização de software; na Seção 3.3 são descritas algumas ferramentas de visualização de software, apresentando as principais características de suas visualizações e metáforas; e na Seção 3.4 são apresentadas as considerações finais com relação ao conteúdo deste capítulo.

3.2 Conceitos de Visualização de Software

A Visualização de Software tem como objetivo facilitar o entendimento de informações estruturais de um software, de modo que elas possam ser mais facilmente compreendidas pelo Engenheiro de Software. Para isso são utilizadas as metáforas visuais, que são formatos de apresentação de conteúdo que combinam elementos visuais, como formas geométricas, grafos e cores, para abstrair informações estruturais do software (OLIVEIRA; LEVKOWITZ, 2003). Como metáforas visuais pode-se citar:

- ***Treemaps*** (SHNEIDERMAN, 1991)**:** representam a estrutura de um software por meio de retângulos aninhados. O retângulo mais externo representa o projeto e os mais internos representam os pacotes, as classes e os métodos desse projeto;
- **Grafo de Dependência:** baseia-se em um grafo (BATTISTA *et al.*, 1994) para representar as informações de dependência do software. Nessa metáfora as classes e métodos de um software são representados pelos nós do grafo e as suas relações de dependências pelas setas direcionadas;

- **Polimétrica:** apresenta os relacionamentos das hierarquias de herança das estruturas de um software por meio de uma estrutura de árvore (BATTISTA *et al.*, 1994). Os nós de uma árvore podem ser representados por figuras geométricas, cujas dimensões também podem representar informações, como exemplo, a largura e o comprimento de um nó podem representar a quantidade de linhas de código fonte e quantidade de métodos de uma classe;

- **Matriz de Acoplamento:** utiliza-se de uma metáfora semelhante a um tabuleiro de batalha naval, que indica as relações de dependências entre os pacotes, classes e métodos de um software. Na primeira coluna do tabuleiro é apresentado o nome da estrutura e um número que a representa, de forma que seja possível relacioná-la com as demais estruturas do software;

- **Hiperbólica** (LAMPING *et al.*, 1995; LAMPING; RAO, 1996)**:** mapeia as classes e métodos de um software como nós, sendo os relacionamentos entre eles representados por arestas direcionadas. Essa visão apresenta um nó em destaque, sendo seus nós "vizinhos" destacados dos demais por meio de uma cor qualquer;

- **Cidades** (WETTEL; LANZA, 2007)**:** determina que as entidades de um software (classes, interfaces e aspectos) sejam representadas como prédios e agrupadas em bairros e distritos, que representam os subpacotes e pacotes, respectivamente;

- **Cores:** podem ser utilizadas para representar diversas informações. Normalmente essa metáfora aparece aliada às demais apresentadas, representando informações como a quantidade de linhas de código fonte ou métodos de uma estrutura do Software. Essa metáfora também é utilizada agrupar as estruturas do software, agrupando estruturas semelhantes por meio de uma mesma cor.

Uma ferramenta de visualização de software tem como objetivo implementar uma ou mais metáforas. É de responsabilidade de uma ferramenta de visualização analisar todo o código fonte do sistema, recuperando as informações necessárias para a criação dos desenhos que representam as metáforas implementadas. A próxima Seção apresenta algumas das ferramentas de visualização presentes na literatura, identificando as metáforas implementadas por elas.

3.3 Ferramentas de Visualização de Software

As ferramentas de visualização de software auxiliam o engenheiro de software a analisar o código-fonte, extrair as informações desejadas e visualizá-las de diferentes formas. Outro

ponto importante de uma ferramenta de visualização é a de permitir a interação do engenheiro de software, que pode refinar e obter diferentes visualizações com o uso de filtros e *zoom*. Essa interação é importante, pois em alguns casos existe o excesso de informação a ser exibida, que sobrecarrega a forma de apresentação dos resultados, dificultando a realização de uma análise correta. Por exemplo, na Figura 3.1 está ilustrada uma visão que utiliza a metáfora de grafo para exibir as relações de dependências entre as classes de um software. Em decorrência da falta de uso de filtros, a quantidade excessiva de setas direcionadas não permite a identificação da origem e do destino uma seta. Na Figura 3.2 é exibido o mesmo conteúdo, porém um filtro é utilizado para exibir somente as relações de dependência de uma classe.

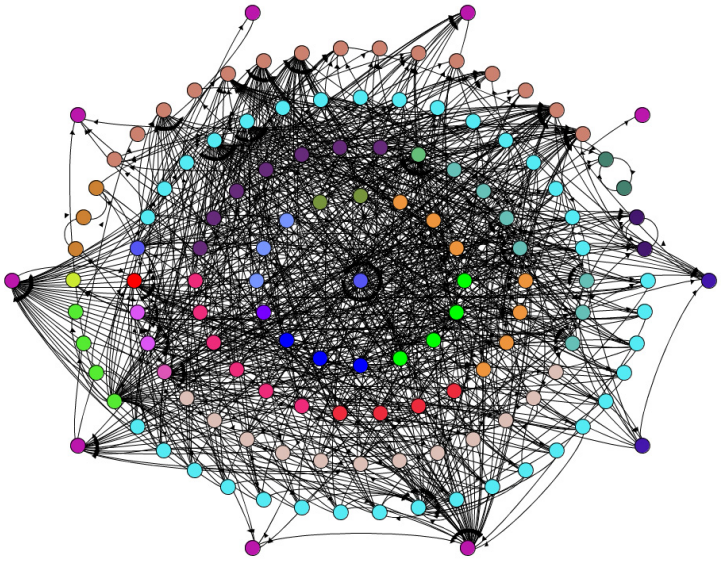


Figura 3.1: Exemplo de visualização sem filtro.

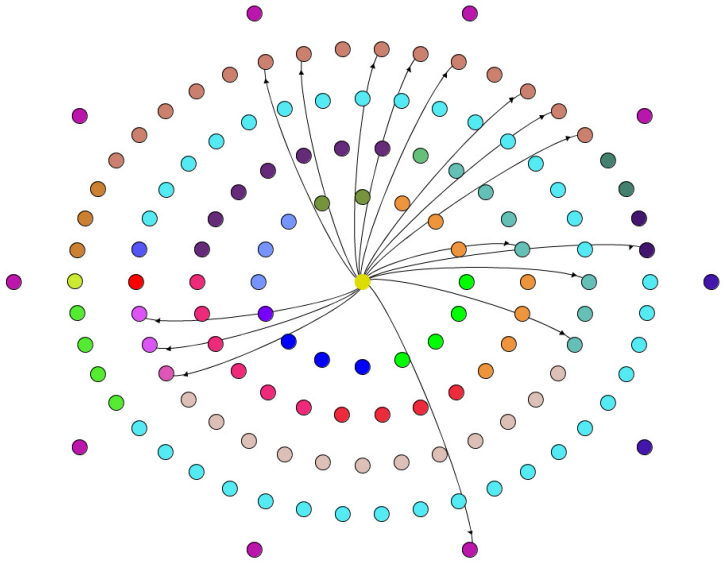


Figura 3.2: Exemplo de visão com filtro.

Nas subseções seguintes são apresentadas algumas das ferramentas de visualização de software presentes na literatura. São apresentadas a descrição de suas principais características e quais as metáforas que são utilizadas para a criação de suas visões.

3.3.1 MosaiCode

MosaicCode (MALETIC *et al.*, 2011) é uma ferramenta para visualização de softwares, desenvolvida em C++, que apresenta quatro visões coordenadas:

1. ***Mosaic View:*** criada com base na metáfora de mosaico, que representa cada entidade do software como uma peça de mosaico, sendo separadas em grupos chamados de *containers*;
2. ***Container:*** mostra a estrutura hierárquica das entidades de um determinado *container*;
3. ***Summary:*** um histograma representando como as entidades estão espalhadas dentro do *container*;
4. ***Entity Views:*** mostra atributos e valores das entidades.

Essa ferramenta é responsável por gerar o ambiente de visualização, sendo responsabilidade de outra ferramenta analisar o código fonte e gerar as informações que serão exibidas. Para que a MosaicCode possa entender as informações identificadas, elas devem ser escritas em um arquivo XML ou CSV, sendo que o XML é baseado no formato da ferramenta sv3d (MARCUS *et al.*, 2003).

Também é suportada a exibição de informações da evolução das versões de um software, que por meio de um conjunto de botões, similar ao de um *player* de vídeo, exibe, de maneira animada, as informações de diversas versões.

Na Figura 3.3 é mostrado um exemplo de utilização da ferramenta MosaicCode extraído de Maletic *et al.* (2011), pois a ferramenta não se encontra disponível para download. Nessa figura são mostradas as quatro visões: *Mosaic* (A), *Container* (B), *Summary* (C) e *Entity* (D). A área indicada pela letra E é responsável pela exibição das informações das versões sendo analisadas, sendo que, à esquerda, consta o nome da versão em análise e, ao centro, estão os botões responsáveis por exibir, de maneira animada, as informações sobre as versões do software.

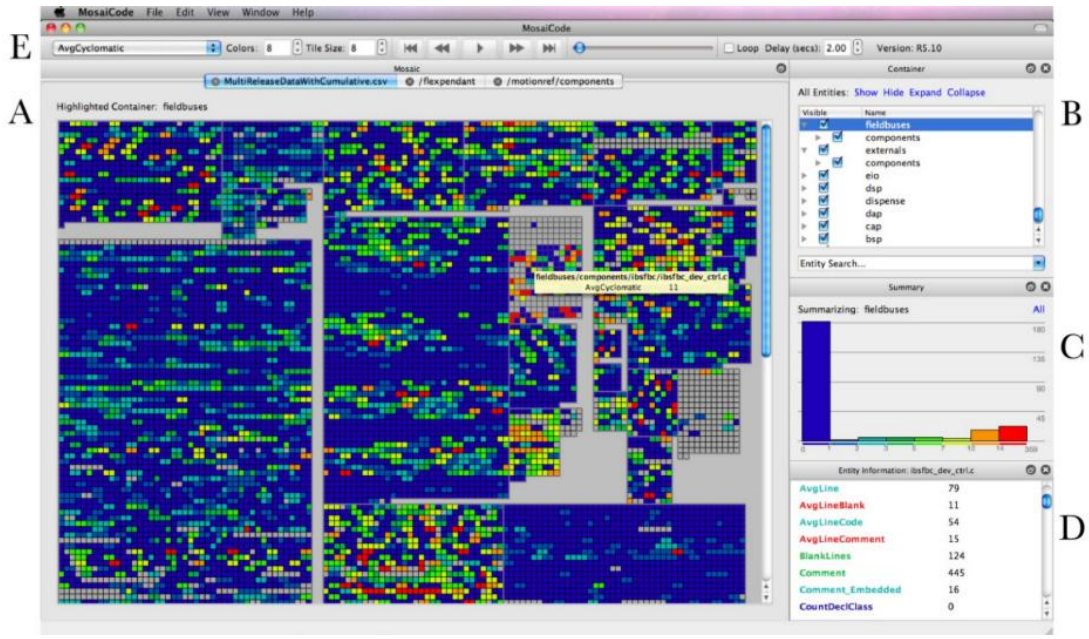


Figura 3.3: Tela da ferramenta MosaiCode, extraída de (MALETIC *et al.*, 2011).

3.3.2 PEVAD

PEVAD (ASPECTOS, 2007) é uma ferramenta de visualização de software orientado a aspectos, que exibe as dependências entre os aspectos e as classes e métodos de um software. Essa ferramenta foi desenvolvida em linguagem Java e pode analisar somente softwares orientados a aspectos escritos em linguagem de programação AspectJ. Por meio da análise dos *bytecodes*, que são os resultados da compilação do código fonte Java de um software, essa ferramenta pode identificar os aspectos e as classes e métodos afetados por ele, montando um grafo de dependência (ASPECTOS, 2007).

O PEVAD possui um ambiente interativo no qual pode-se alterar as cores utilizadas nos grafos de dependências, mover um ou mais vértices do grafo, para facilitar a visualização, e aumentar ou diminuir o tamanho das estruturas que compõe o grafo (*zoom*).

O grafo gerado pelo PEVAD possui três níveis de visualização. O primeiro é responsável por mostrar a dependência entre os aspectos e as classes. Já o segundo é responsável por representar a dependência entre os aspectos e os métodos. Nesses dois níveis, um vértice, que representa um aspecto, é ligado a todos os vértices que representam as classes ou métodos afetados por ele. O terceiro, e último nível, possibilita a visualização de um método afetado por um aspecto, de modo que possa ser identificado qual o momento, do fluxo da execução desse método, que é afetado pelo aspecto. Para facilitar a visualização, um vértice que representa um aspecto é representado, por padrão, pela cor vermelha.

3.3.3 SourceMiner

SourceMiner (CARNEIRO *et al.*, 2010) foi desenvolvida como um plug-in do Eclipse (Eclipse Foundation, 2013) e auxilia a visualização das informações estruturais de um software desenvolvido em Java, fornecendo um ambiente baseado em múltiplas visões. Cada visão é baseada em uma metáfora, complementar a outra, que auxilia melhor na exibição de algumas informações estruturais.

A primeira visão dessa ferramenta é criada com base na metáfora *Treemap* (SHNEIDERMAN, 1991) (Figura 3.4). A coloração inicial dos retângulos representa a quantidade de linhas de código fonte do um método. A coloração mais clara representa menor quantidade de linhas de código. A quantidade de ITs que afetam um método é expressa por um número colocado dentro do retângulo que o representa. Ao deixar o cursor do mouse sobre um retângulo que representa um método, são exibidos detalhes, como nome do pacote, nome da classe, nome do método, quantidade de linhas de código e o nome dos ITs que o afetam. A visão *Treemap* da SourceMiner tem o recurso de *zoom*, com a qual é possível ter a visão nos retângulos mais internos, caso o engenheiro de software deseje analisar melhor um pacote ou uma classe.

Figura 3.4: Exemplo da Visão *Treemap*.

Outra metáfora existente é a Polimétrica, que utiliza retângulos para representar as classes do sistema, cuja largura representa o número de métodos e a altura representa a quantidade de linhas de código dessa classe. A coloração inicial é utilizada para diferenciar as classes

segundo seus tipos, que podem ser: abstratas, concretas, interfaces e externas (provenientes de bibliotecas), como mostrado na Figura 3.5.

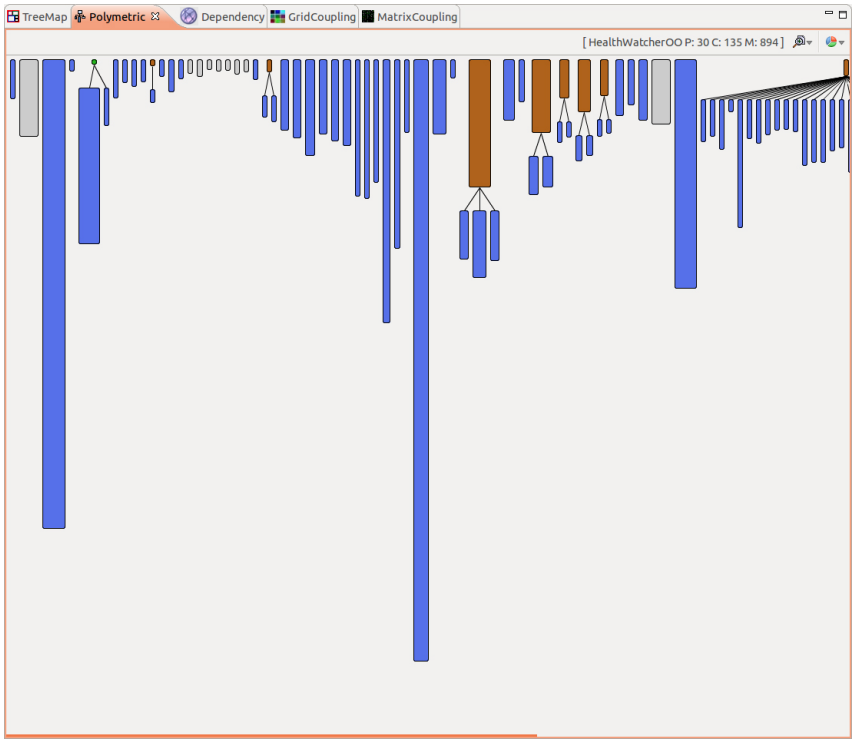


Figura 3.5: Exemplo da Visão Polimétrica.

A Visão de Dependência utiliza-se da metáfora de grafos para exibir as relações de dependência entre os pacotes e as classes de um software, analogamente à metáfora de grafo apresentada na Seção 3.2. A coloração inicial é utilizada para indicar o pacote de cada classe, como mostrado na Figura 3.6, que apresenta as relações de dependência de uma única classe do software.

A quarta metáfora apresentada é similar à de um tabuleiro de xadrez, pois apresenta todas as classes do software como quadrados, ordenados em ordem decrescente de acordo com algum valor significativo. Esse valor pode ser: i) as relações de dependência que uma classe apresenta com as demais; ii) as relações de dependência que as demais classes do software apresentam com uma classe e; iii) a soma dos valores anteriores. A coloração inicial representa o pacote ao qual a classe pertence e um exemplo dessa visão pode ser vista na Figura 3.7. Ao realizar um duplo clique sobre uma classe, é exibido um grafo, similar ao da visão anterior, com todas as dependências dessa classe. Esse grafo apresenta seus nós em uma espiral e um exemplo pode ser visto na Figura 3.8, que foi obtido após um duplo clique sobre o primeiro quadrado da Figura 3.7.

A última visão existente na SourceMiner é a matriz de acoplamento, que pode ser vista na

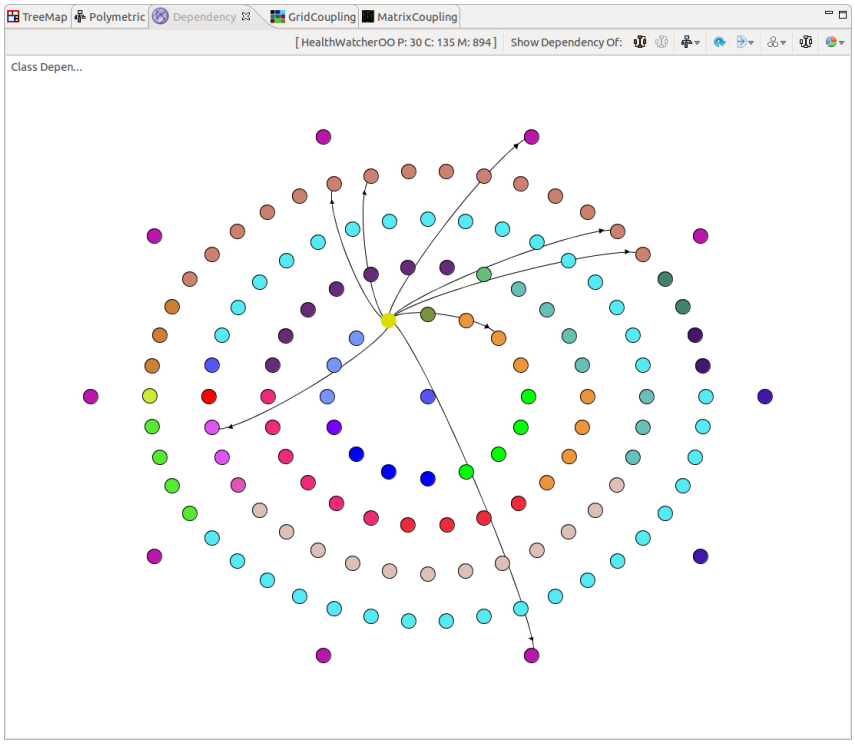


Figura 3.6: Exemplo da Visão de Dependência.

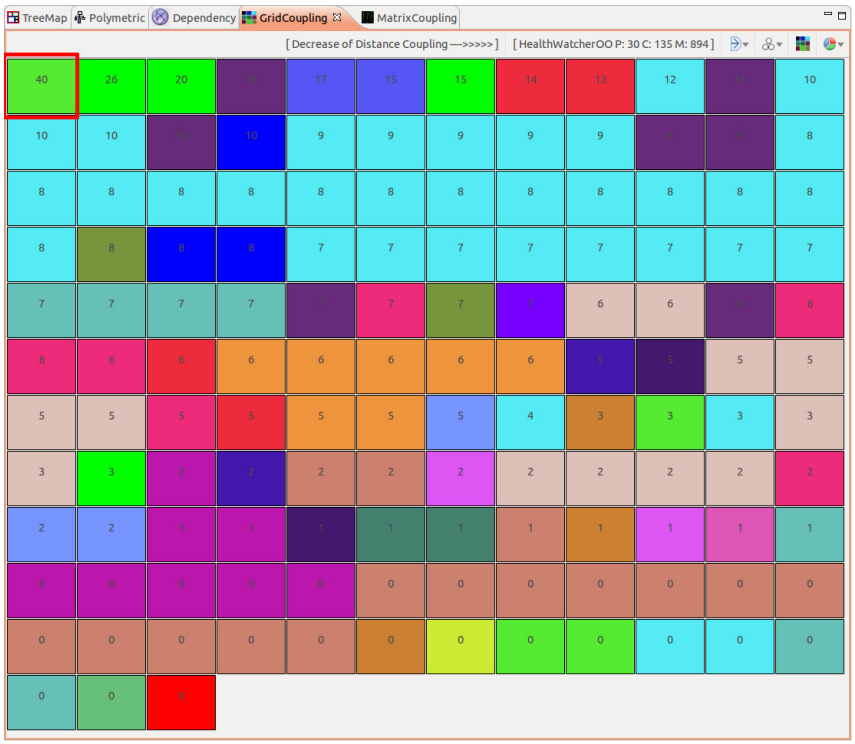


Figura 3.7: Exemplo da Visão de Acoplamento.

Figura 3.9. A cor cinza em uma célula do tabuleiro, apresentado por essa visão, indica que essas estruturas estão relacionadas.

Todas as visões apresentadas permitem a exibição dos indícios de ITs identificados pela

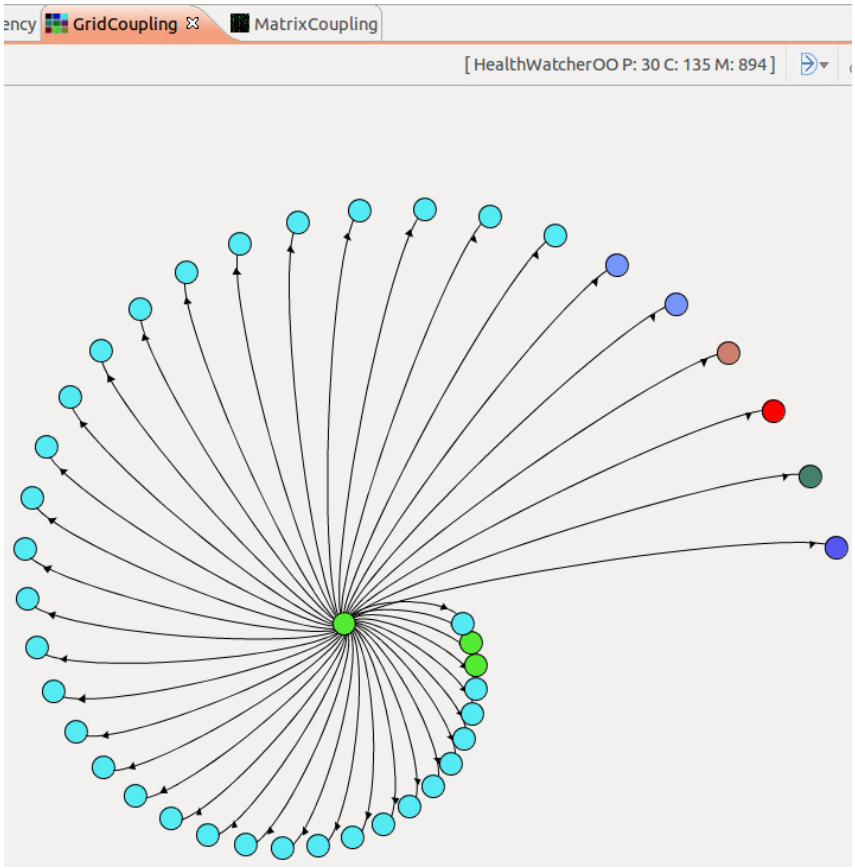


Figura 3.8: Exemplo da Visão Espiral.

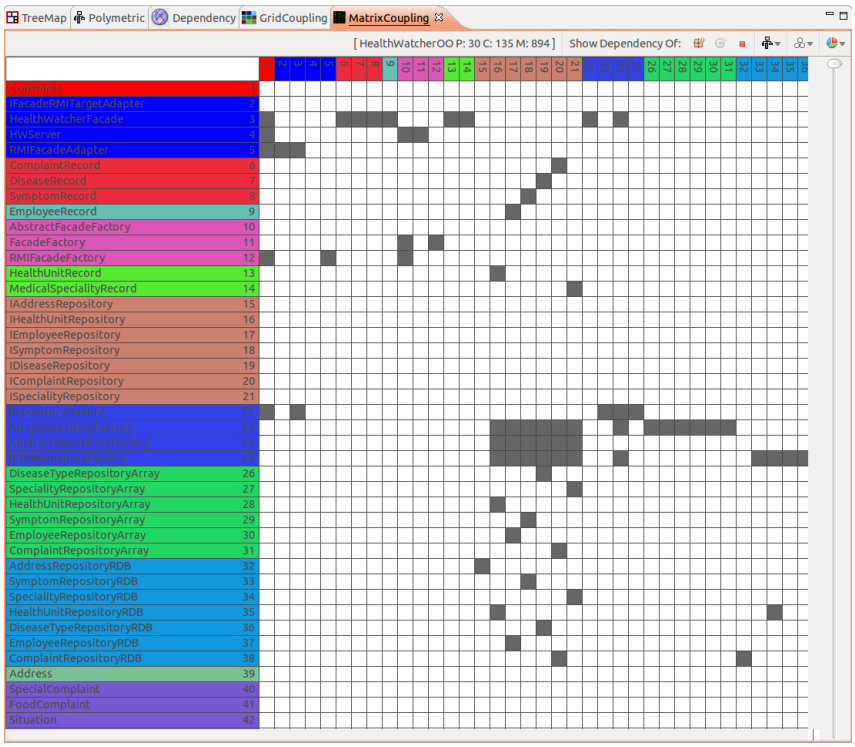


Figura 3.9: Exemplo da Visão Matriz de Acoplamento.

ferramenta de mineração ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005), apresentada no Capítulo 2. Isso ocorre por meio da substituição da coloração inicial, sendo que as estruturas passam a ser coloridas de acordo com o IT que as afetam. Na Figura 3.10, área A, pode ser visto um exemplo da Visão *Treemap*, sendo que alguns indícios do IT de persistência foram identificados pela ferramenta ConcernMapper e coloridos com a cor verde. Na área B, é exibida a aba responsável pela coloração do IT identificado e um *checkbox*, que permite selecionar quais ITs devem ser coloridos.

Figura 3.10: Exemplo da Visão *Treemap* colorida de acordo com alguns indícios do IT de persistência.

Silva A. N.; Carneiro (2012) propuseram uma nova estrutura para estender a funcionalidade da SourceMiner, que divide o código fonte dessa ferramenta em três camadas:

1. **Camada de Modelagem:** define as atividades de importação de dados a serem realizadas. É possível a criação de Módulos de importação específicos para que novos dados sejam incluídos nas visualizações da SourceMiner;

2. **Camada de Controle:** é responsável pelo processamento dos dados importados, controlando a comunicação entre os componentes das demais camadas. Também é responsável pelas atividades de filtragem dos dados importados pela primeira camada, de modo que por meio de regras definidas em seus módulos as informações a serem exibidas na próxima camada são selecionadas;

3. **Camada de Visualização:** responsável pelo controle das visões apresentadas pela ferramenta. Assim, é possível inserir novas visões, utilizando novas metáforas para exibir os dados importados.

Cada camada pode ser composta por um ou mais *plug-ins* do Eclipse o que permite inserir/remover funcionalidades com a inserção/remoção de *plug-ins*, respectivamente.

3.3.4 $SoftVis_{4CA}$

$SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013), anteriormente denominada SoftVis4AspectMining, é uma ferramenta de visualização que analisa softwares desenvolvidos em linguagem de programação Java. Essa ferramenta disponibiliza um ambiente capaz de analisar visualmente os resultados apresentados por duas técnicas de mineração de interesses transversais: Fan-in (MARIN *et al.*, 2007) e Program Slicing (KATTI *et al.*, 2012). Essa ferramenta disponibiliza seis visões coordenadas, detalhadas a seguir.

A Visão Inter-Classes e a Visão Inter-Métodos permitem a visualização das chamadas entre métodos em nível de classe e de método, respectivamente. As duas visões utilizam a metáfora hiperbólica, cujos exemplos são apresentados nas Figuras 3.11 e 3.12. A $SoftVis_{4CA}$ é capaz de calcular o valor Fan-in dos métodos de um software e exibi-los nesta visão, por meio de uma escala de cores, sendo que o cinza representa baixo valor Fan-in e o vermelho alto.

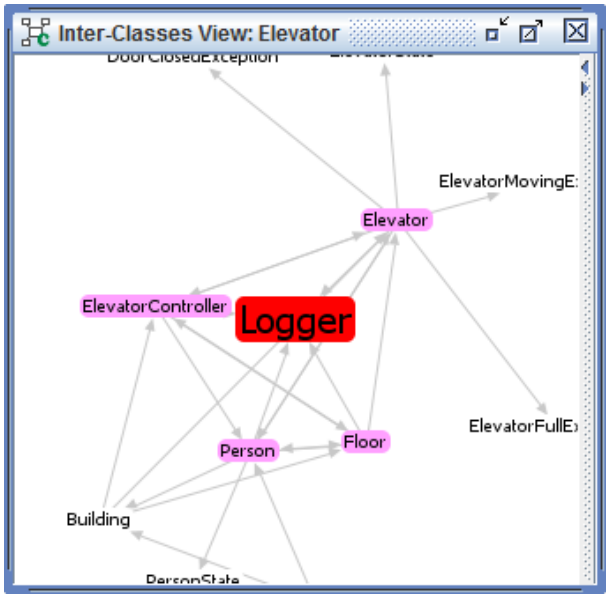


Figura 3.11: Exemplo da Visão Inter-Classes, extraído de (DELFIM, 2013).

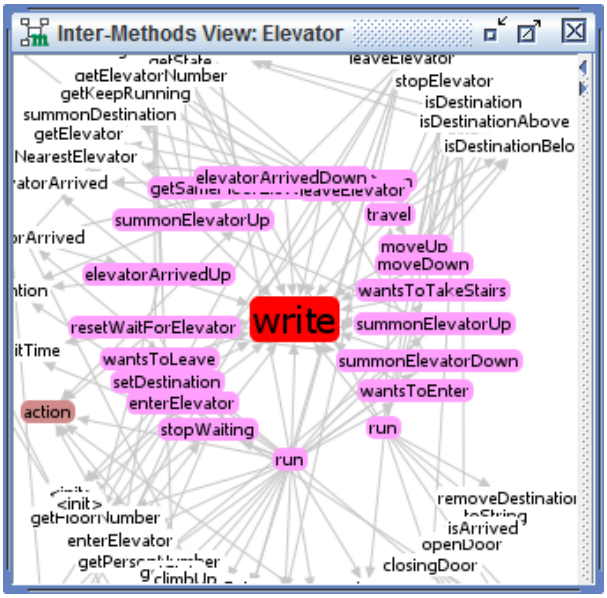
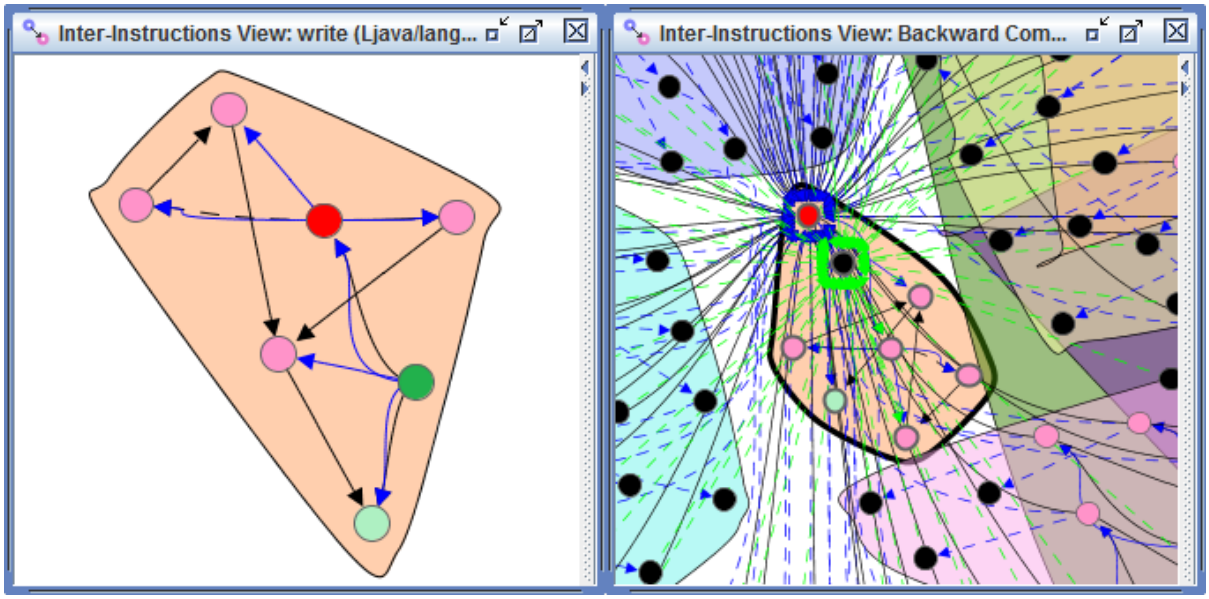


Figura 3.12: Exemplo da Visão Inter-Métodos, extraído de (DELFIM, 2013).

As Visões Inter-Instruções apresentam grafos em nível de instruções, que podem ser intra-método (apresenta o grafo de um método) ou inter-métodos (apresenta grafos de métodos combinados por chamadas de métodos), respectivamente apresentados nas Figuras 3.13(a) e 3.13(b). Nessa visão os conjuntos de instruções *bytecode*, as entradas e saídas de métodos e os parâmetros, são mapeados em nós e agrupados por polígonos coloridos que representam o método ao

qual pertencem. As setas direcionadas representam o fluxo de controle, a dependência de controle e a dependência de dados, sendo diferenciadas por suas cores. Um exemplo é apresentado na Figura 3.13



(a) Grafo do método `write` (Intra-Método).

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

Figura 3.13: Exemplos das Visões Inter-Instruções, extraídos de (DELFIM, 2013).

A Visão Estrutural utiliza a metáfora *Treemap* (SHNEIDERMAN, 1991) para a apresentação da organização estrutural do programa, análoga à apresentada pela SourceMiner, tendo as seguintes diferenças: i) a apresentação de classes aninhadas (*inner classes*); e ii) o tamanho dos retângulos que representam os métodos é proporcional à sua quantidade de instruções *bytecode*. Essa visão apresenta os relacionamentos de um método selecionado (cor vermelha) por meio da coloração rosa, o que permite a análise do impacto causado ao se alterar um método. Um exemplo dessa visão pode ser visto na Figura 3.14.

A Visão de Distribuição de Instruções (Figura 3.15) apresenta as classes do software como barras, sendo suas alturas proporcionais à quantidade de instruções *bytecode* das classes. As listras representam as instruções do programa que compõem fatias obtidas por meio da técnica *Program Slicing*, sendo que cada fatia é representada por uma cor diferente.

A Visão de Bytecode exibe as instruções *bytecode* de um software usando duas barras de rolagem sincronizadas. Na barra de rolagem do lado direito são exibidos conjuntos de instruções, proporcionando uma visão geral das instruções presentes no *bytecode*. Ao selecionar um conjunto de instruções, são exibidas, na barra de rolagem do lado esquerdo, as instruções que ele representa. Um exemplo dessa visão é apresentado na Figura 3.16.

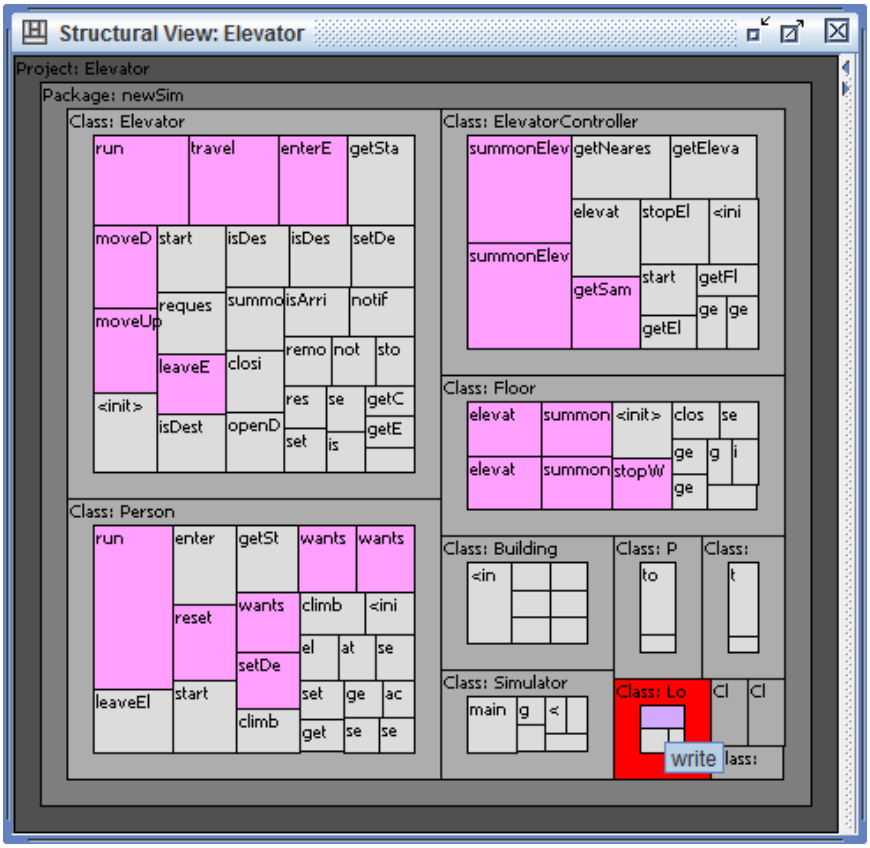


Figura 3.14: Exemplo da *Visão Estrutural*, extraído de (DELFIM, 2013).

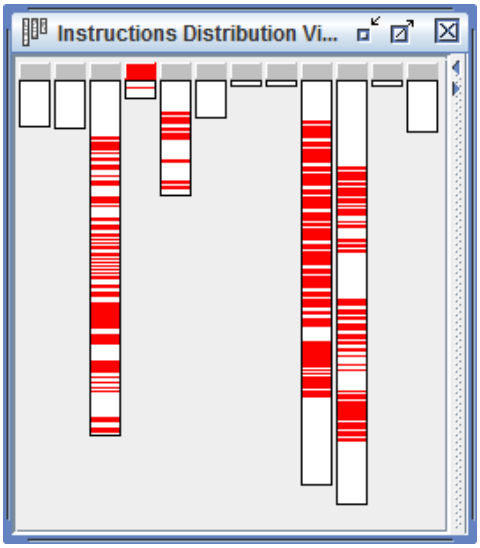


Figura 3.15: Exemplo da *Visão de Distribuição de Instruções*, extraído de (DELFIM, 2013).

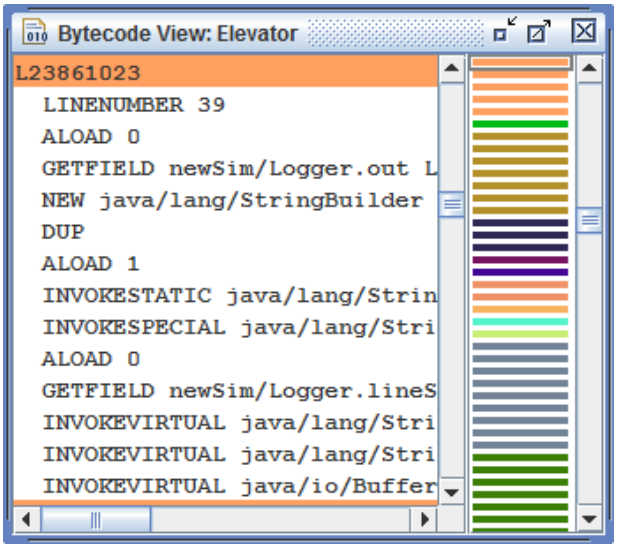


Figura 3.16: Exemplo da *Visão de Bytecode*, extraído de (DELFIM, 2013).

3.3.5 VizzAspectJ-3D

VizzAspectJ-3D (BENTRAD; MESLATI, 2011) é uma ferramenta de visualização de software que, por meio da metáfora de cidades (WETTEL; LANZA, 2007), apresenta as informações estruturais de softwares Orientados a Aspectos e implementados com a linguagem AspectJ.

As informações de um software são exibidas por meio de duas visões baseadas nessa metáfora, uma chamada “VizzJava City”, que exibe as informações da parte Orientada a Objetos do software e outra chamada “VizzAspectJ City” que exibe as informações da parte Orientada a Aspectos. A própria VizzAspectJ-3D, após análise do software, recupera informações estruturais como: i) número de métodos, atributos e linhas de código; ii) relações de hierarquias dos pacotes, classes e aspectos e; iii) relações de dependência entre as classes e os aspectos. Essa ferramenta também apresenta o resultado de um conjunto de nove métricas consideradas importantes pelos seus autores para a análise de softwares OA, como: número de métodos e atributos declarados por uma classe e número de métodos, atributos, pontos de corte e adendos declarados por um aspecto. Essas informações são utilizadas para construir a cidade. Na Figura 3.17 são apresentadas as duas visões, “VizzJava City” e “VizzAspectJ City”, com as informações do software AJHotDraw (AMR, 2013).

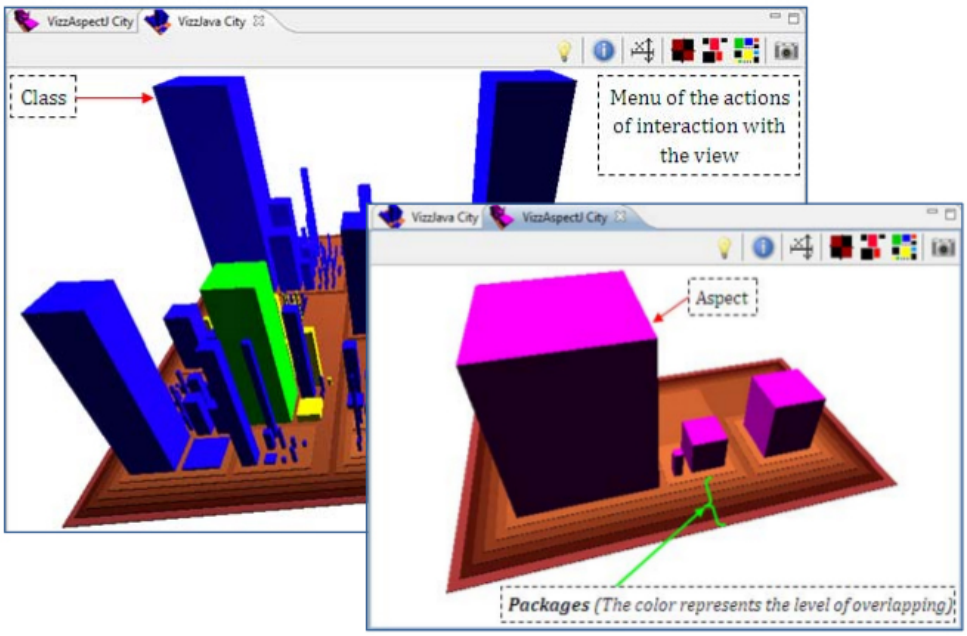


Figura 3.17: Tela da ferramenta VizzAspectJ-3D, extraída de (BENTRAD; MESLATI, 2011).

3.4 Considerações Finais

As visões das ferramentas de visualização de software, por meio de suas metáforas, apresentam de uma forma geral as informações estruturais de um software. Uma metáfora, nem sempre pode exibir todas as informações, necessitando de visões complementares, como pode ser visto na Tabela 3.1. Na primeira coluna da tabela estão listadas as ferramentas apresentadas neste capítulo. Na segunda coluna é apresentada a quantidade de visões que a ferramenta disponibiliza e, na terceira coluna é informada a quantidade de metáforas implementadas pela

Tabela 3.1: Número de Visões e Metáforas das ferramentas apresentadas.

	Visões	Metáforas
MosaiCode	4	1
PEVAD	1	1
SourceMiner	6	6
$SoftVis_{4CA}$	6	5
VizzAspectJ-3D	2	1

ferramenta.

Algumas das técnicas de mineração de ITs apresentadas no Capitulo 2 são complementares, uma vez que a comparação de seus resultados pode auxiliar na identificação de falsos negativos e falsos positivos. As diversas visões que as ferramentas de visualização podem apresentar são uma forma de combinar e analisar esses resultados. Essas visões auxiliam na comparação dos resultados e evidenciam padrões visuais que indicam indícios com maior probabilidade de serem falsos positivos ou falsos negativos. Para possibilitar essa combinação é necessário integrar essas ferramentas. Contudo, essa integração não é trivial, pois não existe uma forma de comunicação padrão entre elas e, normalmente, as ferramentas de visualização não disponibilizam uma forma de integração, necessitando que sejam realizadas alterações em seus códigos fonte. A fim de que essa atividade de integração seja concretizada, este trabalho propõe um metamodelo para a identificação dos indícios de ITs identificados no código fonte, sendo utilizado como forma de comunicação entre essas ferramentas. Esse metamodelo é apresentado no próximo capítulo.

Capítulo 4

METAMODELO PROPOSTO

4.1 Considerações Iniciais

Para auxiliar na atividade Mineração de Interesses Transversais (MIT) existem várias ferramentas, tais como: i) ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005), ii) ComSCId (PARREIRA JÚNIOR *et al.*, 2010a; PARREIRA JÚNIOR *et al.*, 2010b); iii) Fint (MARIN *et al.*, 2007); iv) XScan (NGUYEN *et al.*, 2011); v) ccfinder (KAMIYA *et al.*, 2002) e; FEAT (ROBILLARD; MURPHY, 2003; ROBILLARD; MURPHY, 2007). Essas ferramentas visam a auxiliar o engenheiro de software na busca por indícios de ITs presentes no código fonte de um software e possuem algum tipo de visualização que permita a exibição dos resultados obtidos, porém não tem o enfoque de criar diversas formas de visualização para esses resultados. Muitas vezes essas visualizações são simplesmente arquivos texto, como apresentado pela ferramenta XScan, ou uma tela que apresenta uma lista dos indícios encontrados, como as ferramentas ConcernMapper, FINT e ComSCId.

Ceccato *et al.* (2006) mostraram as vantagens da utilização combinada de diversas ferramentas de MIT, ao apresentarem três diferentes ferramentas para identificar os ITs existentes no sistema JHotDraw (JHOTDRAW.ORG, 2013), um *framework* de editor gráfico *opensource*. Os autores concluíram que, quando são utilizadas diversas ferramentas em um mesmo sistema, as limitações de cada uma delas tornam-se evidentes, aumentando a percepção de possíveis falsos negativos, já que indícios que não foram identificados por uma ferramenta possivelmente foram identificados por outra. Os autores também citaram que pretendiam combinar outras ferramentas, a fim de alcançar melhores resultados por meio dessa combinação. Nenhum ambiente foi mencionado como sendo utilizado para agrupar ou visualizar conjuntamente os resultados apresentados pelas três ferramentas, sendo o trabalho de comparação realizado manualmente.

Algumas das ferramentas de visualização existentes são: i) SourceMiner (SILVA A. N.; CARNEIRO, 2012; CARNEIRO *et al.*, 2010); ii) $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013); iii) MosaicCode (MALETIC *et al.*, 2011) iv) PEVAD (ASPECTOS, 2007) e; v) VizzAspectJ-3D (BENTRAD; MESLATI, 2011). Essas ferramentas visam, por meio de suas visualizações e metáforas, a abstrair as informações estruturais de um software, auxiliando o engenheiro de software a descobrir padrões "escondidos" dentro do código fonte. Como exemplos desses padrões tem-se os *code smells* (FOWLER *et al.*, 1999), que indicam anomalias na modularização das classes de um sistema. Dentre as ferramentas citadas, algumas mostram a necessidade de visualizar as informações referentes aos ITs por meio de suas visualizações como: SourceMiner, $SoftVis_{4CA}$, VizzAspectJ-3D e PEVAD, porém, somente a SourceMiner possui uma forma de integração com as ferramentas de MIT. Para as ferramentas $SoftVis_{4CA}$, VizzAspectJ-3D e PEVAD, todas as informações referentes aos ITs são encontradas por elas mesmas, sendo que as atividades de mineração dessas informações estão programadas em seus próprios códigos fonte. Dessa forma, há dificuldade em integrar novas ferramentas de MIT e para que novas informações sejam exibidas.

Já a ferramenta SourceMiner tem a preocupação em fornecer uma interface para a integração. Em sua primeira versão (CARNEIRO *et al.*, 2010), a integração com as ferramentas de MIT ocorre por meio da troca de arquivos com a ferramenta ConcernMapper. Com o objetivo de possibilitar que mais ferramentas, não só de MIT, fossem integradas a ela, foi desenvolvida uma segunda versão (SILVA A. N.; CARNEIRO, 2012). Nessa versão foi implementada uma nova estrutura que aumenta a sua capacidade de integração, tanto em dados a serem importados quanto em visualizações a serem utilizadas para exibir os dados importados.

Para se obter resultados mais satisfatórios na mineração de aspectos é interessante utilizar ferramentas tanto de visualização quanto de mineração, sendo assim há a necessidade de integrá-las. Entretanto, essa integração não é um processo trivial. Em alguns casos, são necessárias alterações no código fonte das ferramentas, por exemplo, como ocorre com a $SoftVis_{4CA}$, a VizzAspectJ-3D e a PEVAD. Em outros casos é necessário o entendimento de formatos de arquivos que armazenam o resultado obtido. Geralmente, o formato desses arquivos não é trivial e há pouca documentação sobre eles, como ocorre com a SourceMiner e o formato do arquivo gerado pela ferramenta ConcernMapper (ROBILLARD, 2013; ROBILLARD; WEIGAND-WARR, 2005). Ainda há o caso da segunda versão da SourceMiner, que apesar de facilitar a integração de novas ferramentas por meio de sua estrutura, há a necessidade de se adequar às diferenças das ferramentas de MIT apresentadas, já que geram arquivos com formatos diferentes.

No intuito de se utilizar ferramentas de visualização de software juntamente com ferramentas de MIT de forma mais fácil, um metamodelo foi criado neste projeto, chamado *Indicantions Metamodel* (IMM) (TANNER *et al.*, 2013), que serve como uma forma de comunicação entre as ferramentas de MIT e as de visualização de software. O enfoque desse metamodelo é padronizar a forma como as informações dos indícios de ITs, encontradas pelas ferramentas de MIT, devem ser informadas às ferramentas de visualização. Com ele, é possível agrupar diversas ferramentas de MIT e, assim, apresentar diferentes formas de visualização das informações obtidas. Com a integração de ferramentas de MIT com as ferramentas de visualização de software, surgem alguns padrões visuais que podem auxiliar na identificação de falsos positivos e falsos negativos apresentados por uma ferramenta de MIT. Por exemplo, considere uma classe que apresenta somente um método afetado por um IT, sendo que as demais classes pertencentes ao mesmo pacote não apresentam métodos afetados. Neste caso, há uma grande probabilidade desse método ser um falso positivo. Considere também o caso contrário: uma classe que não possui métodos afetados, sendo que todas as demais classes desse pacote apresentam ao menos um método afetado por um mesmo IT. Neste caso, há a possibilidade de que em algum dos métodos dessa classe exista um falso negativo. A identificação desses padrões visuais é facilitada devido à integração dessas ferramentas, pois é possível comparar os resultados obtidos com as diferentes ferramentas de MIT por meio das visões das ferramentas de visualização de software.

Na Seção 4.2 será apresentado o processo de criação do IMM, mostrando metamodelos relacionados que contribuíram nesse processo. Na Seção 4.3 será detalhado o metamodelo programado, juntamente com suas classes e funções de auxílio de utilização. Por fim, na Seção 4.4, serão apresentadas as considerações finais.

4.2 Criação do Metamodelo

O intuito da criação do metamodelo foi a possibilidade de utilizar diferentes ferramentas de MIT em conjunto com as de visualização de software, de forma que os resultados pudessem ser combinados, a fim de obter informações mais completas a respeito de um software.

Na literatura há diversos metamodelos para identificar os ITs e as estruturas do sistema afetadas por eles. PARREIRA JÚNIOR *et al.* (2010a) propuseram um metamodelo para definição de modelos de classes OO anotados com indícios de ITs. Esse metamodelo define que as classes dos modelos criados devem possuir estereótipos para indicar os indícios de ITs identificados pela ferramenta ComSCId. O metamodelo construído por PARREIRA JÚNIOR *et al.* (2010a) não pode ser utilizado neste projeto de integração de ferramentas MIT e de visualização, pois

esse metamodelo há mais informações do que as necessárias para a identificação de um indício de IT no código fonte, tais como informações de associação entre as classes do sistema. Esse tipo de informação não é necessária neste projeto, sendo que as ferramentas de visualização conhecem a estrutura do sistema, e assim é necessário somente indicar as estruturas afetadas.

Piefel (2006) propõe um metamodelo chamado *CeeJay*, para a geração de código fonte, que possibilita a geração tanto de código C++, quanto Java. No escopo deste trabalho, esse metamodelo poderia ser utilizado como uma forma de representar o código fonte, porém não apresenta nenhuma informação quanto aos ITs e nem quanto as estruturas afetadas.

Bernardi e Lucca (2011) propõem um metamodelo capaz de representar a estrutura estática de sistemas OO de acordo com os ITs que a afetam. Esse também apresenta muitas informações não relevantes para este projeto, como o relacionamento entre os módulos de um sistema, relações entre classes, entre métodos (chamados e chamadores) e heranças.

Então, o metamodelo proposto por PARREIRA JÚNIOR *et al.* (2010a) foi considerado como base para a criação do *Indications Metamodel* (IMM), por ter sido desenvolvido no mesmo grupo de pesquisa do autor dessa dissertação e pelo seu conhecimento sobre o mesmo.

O metamodelo IMM (Figura 4.1) foi elaborado de modo que possa ser identificada a ferramenta e a técnica de MIT utilizadas no projeto em análise. Para essa finalidade foi criada a classe `MiningTool`, que armazena o nome (`name`) da ferramenta e o nome da técnica (`miningMethod`) que foram utilizadas para identificar os indícios de IT representados pelo IMM. Essa classe é específica do domínio da MIT, porém, devido a simplicidade do IMM, focado em representar o código fonte de uma aplicação, pode-se alterá-lo, adicionando ou removendo classes, de forma que o IMM possa se adequar a diferentes domínios.

A classe `MiningTool` está relacionada a um projeto (`Project`), que tem como atributos seu nome (`name`) e caminho do sistema (`path`). Um projeto está associado a diversos pacotes (`Package`) que são identificados por meio de seus nomes (`name`).

Um pacote pode conter diversas classes (`Class`), que têm como atributos `id` e `name` para que possam ser diferenciadas entre si. O `id` de uma classe normalmente tem como padrão a combinação entre o nome do pacote e o nome da classe, separados pelo caractere de ponto final (“.”). Uma classe pode conter diversas classes internas, as quais, por sua vez, podem conter outras classes internas. Para possibilitar a representação dessas classes internas, foi necessário que o IMM tivesse o relacionamento de composição recursivo. O `id` de uma classe interna normalmente tem como padrão a junção do `id` da classe externa com o nome da classe interna separados pelo caractere de ponto final.

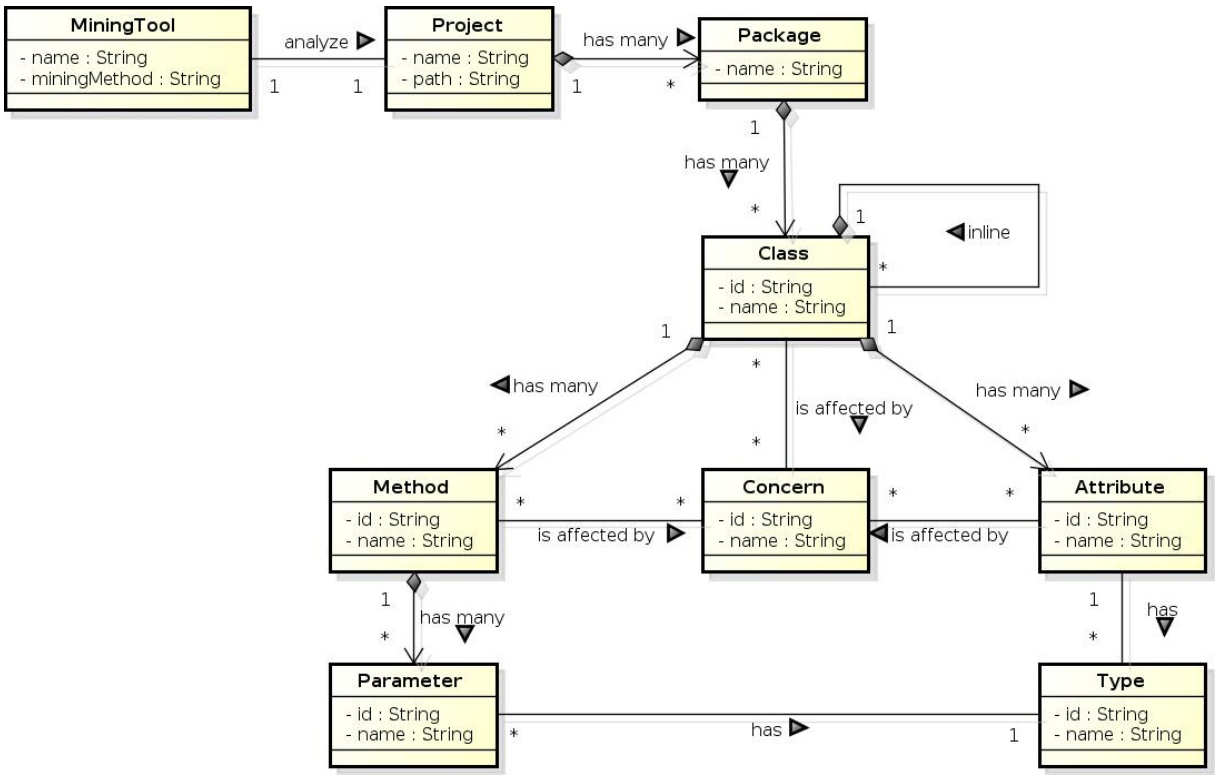


Figura 4.1: ***Indications Metamodel*.**

Uma `Class` pode conter diversos métodos (`Method`), que podem conter diversos parâmetros (`Parameter`). Para a identificação de um método não há a necessidade de saber qual o seu tipo de retorno, visto que nenhuma linguagem OO aceita dois métodos com o mesmo nome e os mesmos parâmetros, porém com retornos diferentes. Sendo assim, uma segunda atualização foi realizada no IMM, na qual o relacionamento entre a classe `Method` e a classe `Type` foi removido. O `id` de um método é normalmente formado pelo `id` da classe, combinado com o caractere ponto final, seguido do nome do método, novamente do caractere ponto final e do nome dos tipos de todos os seus parâmetros, separados pelo caractere ponto final. Os tipos dos parâmetros de um método são essenciais para o reconhecimento de métodos sobrecarregados, visto que possuem o mesmo nome. O tipo de um parâmetro é representado pela classe `Type`. O `id` de um parâmetro normalmente segue como padrão a junção do `id` do método com o nome do parâmetro, separados pelo caractere ponto final. Não há a necessidade de armazenar se um tipo é uma classe, ou um tipo primitivo, somente um conjunto de caracteres com o nome do tipo é suficiente, por isso essa classe não apresenta nenhuma relação com a classe `Class`.

Uma `Class` também pode conter diversos atributos (`Attribute`), sendo que cada atributo deve possuir um tipo (`Type`). A relação entre atributo e tipo poderia não existir, visto que nenhuma linguagem de programação OO aceita dois atributos de uma mesma classe com um mesmo nome, mas com tipos diferentes. Porém, optou-se por deixar essa relação, já que no

processo de análise das informações contidas no metamodelo é interessante saber também o tipo do parâmetro e não somente o seu nome. O `id` de um atributo normalmente é formado pelo `id` da classe combinado com seu nome, separados pelo caractere ponto final.

Toda `Class`, `Method` e `Attribute` do metamodelo estão associados a um interesse (`Concern`). Esse tipo de relação evita que informações desnecessárias, como classe, métodos e atributos não afetados, sejam armazenadas pelo modelo gerado. Porém, caso o desenvolvedor tenha a necessidade de armazenar alguma dessas informações, o metamodelo proposto não o impedirá. O `id` de um interesse não segue nenhum padrão, sendo definido pela ferramenta de mineração.

A relação entre classe e interesse também poderia não existir, pois a relação de uma classe com um determinado interesse poderia ser identificada pela análise de seus métodos e atributos. Optou-se por fazer essa relação para que ferramentas de visualização, que tenham visões cuja granularidade seja em nível de classe, não precisem analisar informações como métodos e atributos, diminuindo o tempo necessário para identificar as classes afetadas.

As classes do IMM, bem como as funções necessárias para a sua utilização, foram implementadas como um *plug-in* do ambiente de desenvolvimento Eclipse (Eclipse Foundation, 2013). A descrição dessa implementação é tratada na próxima seção.

4.3 Metamodelo Programado

A implementação do IMM foi feita como um *plug-in* do Eclipse e utilizada a linguagem de implementação Java. As justificativas para essa escolha são: i) o conhecimento prévio do autor sobre a linguagem Java; ii) um *plug-in* do Eclipse pode ser incluído em qualquer tipo de projeto Java, visto que é um arquivo JAR, ou seja, um arquivo que agrega várias classes Java e; iii) porque muitas ferramentas de MIT e de visualização de software existentes na literatura são também *plug-ins* do Eclipse.

A estrutura de classes do *plug-in* do IMM implementado é apresentada na Figura 4.2. Essa estrutura é composta por seis pacotes, dentre eles o pacote `metamodel`, que contém a classe `Activator`, necessária para todos os projetos que definem *plug-ins* do Eclipse. Essa classe é responsável por gerenciar as regras de ativação do *plug-in* durante a inicialização do Eclipse e não sofreu alteração do padrão gerado ao criar o projeto do *plug-in*. Os demais pacotes e suas respectivas classes e funcionalidades serão explicados detalhadamente nas próximas Subseções. Todos os demais pacotes possuem o prefixo `br.ufscar.dc.metamodel`, que será omitido a partir deste momento.

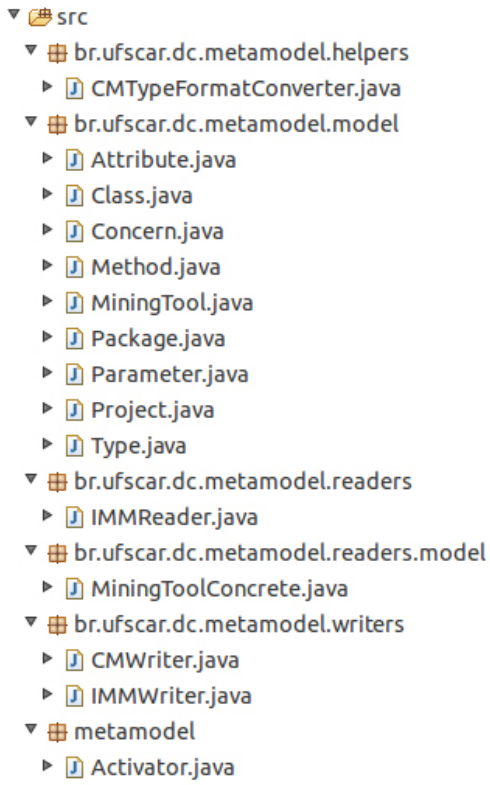


Figura 4.2: Estrutura das classes do metamodelo implementado em linguagem Java.

4.3.1 Pacote `model`

O *Indications Metamodel* foi desenvolvido para ser genérico e conciso, contendo somente as informações necessárias para a identificação dos indícios de ITs encontrados no código fonte. Porém, se houver a necessidade de que mais informações sejam incluídas, esse metamodelo pode ser estendido, sem que o código já existente seja alterado. Dessa forma, as classes do metamodelo podem se adaptar aos diferentes tipos de informações que desejem representar.

O pacote `model` implementa as classes do metamodelo, bem como seus relacionamentos. Todas as classes do metamodelo possuem os métodos acessores para seus atributos, usados de maneira tradicional. A única classe abstrata é a `MiningTool`. Isso se justifica, pois para possibilitar a extensão das demais classes do metamodelo sem afetar as funções de escrita e leitura, houve a necessidade de criar funções capazes de informar a classe `MiningTool` se alguma das demais classes foi estendida. Essas funções são abstratas e chamam-se: i) `getPackageClass`; ii) `getClassClass`; iii) `getMethodClass`; iv) `getAttributeClass`; v) `getConcernClass`; vi) `getParameterClass`; vii) `getProjectClass` e; viii) `getTypeClass`. Uma referência para as classes Java que implementam as classes do metamodelo é retornada por essas funções, podendo retornar classes que foram ou não decoradas. As atividades de escrita e leitura são melhor detalhas nas Subseções 4.3.2 e 4.3.3, respectivamente.

Quando um desenvolvedor for utilizar o metamodelo somente há a necessidade de implementar a classe `MiningTool` quando necessitar das funções de escrita, independente de decorar ou não as classes. Para as funções de leitura, é preciso somente implementar a classe `MiningTool` caso alguma das classes tenha sido decorada no momento da escrita.

A classe `Project`, para auxiliar na atividade de leitura das informações do metamodelo, possui a função `getConcerns` que analisa todas as classes de todos os pacotes do projeto e retorna uma lista com todos os interesses (`Concerns`) distintos que afetam esse projeto. Essa classe também possui a função `addPackage` responsável por verificar se o novo pacote já existe na lista de pacotes de um projeto. Caso não exista, o pacote é incluído e, caso exista, é solicitada a inserção de todas as classes desse pacote no pacote já existente.

A classe `Package` possui a função `addClass` responsável por verificar se a nova classe já existe na lista de classes de um pacote. Caso não exista ela é incluída e, caso exista, é solicitada a inclusão de todos os métodos, atributos e interesses na classe já existente.

A classe `Class` possui as funções `addMethod`, `addAttribute` e `addConcern`. Essas funções são responsáveis por verificar se os métodos, atributos e interesses já estão ou não presentes nas informações dessa classe, e incluí-los quando não presentes. Caso um interesse esteja presente na lista de interesses de uma classe nada é feito, pois já existe a informação de que a classe é afetada por esse interesse. Para atributos e métodos já existentes é somente solicitada a atualização dos interesses que os afetam. Caso os métodos e atributos já tenham relação com um determinado interesse, nada é feito, analogamente ao que acontece com a atualização de um interesse de uma classe. As classes `Class`, `Method` e `Attribute` possuem a função `isAffectedByConcern`, que retorna verdadeiro se a classe, método ou atributo for afetada por um determinado interesse passado como parâmetro, ou falso caso contrário.

4.3.2 Pacote `writers` e `helper`

O pacote `writers` possui duas classes, a `IMMWriter` e a `CMWriter`, responsáveis por gerar um arquivo com todas as informações presentes nos modelos gerados a partir do modelo IMM, em formato IMM e em formato CM, respectivamente. O formato IMM é a serialização em formato XML das classes implementadas do IMM. Essa serialização é feita com o auxílio da biblioteca XStream (XSTREAM, 2013). O formato CM é utilizado como formato de saída da ferramenta de mineração ConcernMapper e foi implementado para manter compatibilidade com versões da SourceMiner que não são capazes de ler os arquivos IMM e também para facilitar o processo de integração com a versão da SourceMiner proposta por Carneiro *et al.* (2010). As atividades realizadas para essa integração são descritas no próximo Capítulo. As classes

`IMMWriter` e `CMWriter` são detalhadas nas subseções 4.3.2.1 e 4.3.2.2, respectivamente. Na subseção 4.3.2.3 é apresentada a única classe do pacote `helpers` (`CMTypeFormatConverter`), que utiliza o conhecimento adquirido na escrita do arquivo CM para criar um conversor que auxilia na leitura desse mesmo formato.

4.3.2.1 Classe `IMMWriter`

A classe `IMMWriter` é responsável por, a partir de um objeto da classe `MiningTool`, gerar um arquivo em formato IMM na raiz do projeto ao qual os indícios pertencem. Esse arquivo é gerado somente se os atributos da classe `MiningTool` possuírem algum valor. O caminho de sistema do projeto no qual o arquivo será gerado é identificado por meio do atributo `path` da classe `Project`. O nome do arquivo gerado, por padrão, é o nome da ferramenta de mineração, obtido por meio do atributo `name` da classe `MiningTool`, sendo que todos os caracteres de espaço são trocados pelo caractere *underline* ("_"), com a extensão ".imm". As funções da classe `MiningTool`, responsáveis por indicar se alguma classe do metamodelo foi decorada, são utilizadas nesta classe para indicar à biblioteca XStream as classes que devem ser serializadas, fazendo com que toda informação adicional seja escrita nos arquivos.

4.3.2.2 Classe `CMWriter`

O formato CM é baseado no padrão Java de conversão de classes, métodos e atributos para cadeias de caracteres (`Strings`), porém apresenta diversas particularidades. Esse formato é de difícil compreensão e pouco documentado (ROBILLARD, 2013; ROBILLARD; WEIGAND-WARR, 2005). As informações aqui apresentadas foram identificadas pelo autor desta dissertação ao longo de sua experiência nas atividades de leitura e escrita de arquivos CM. Todo arquivo CM possui o formato XML, sendo a *tag* raiz representada pela *tag* `<model>`. Essa *tag* é responsável por agrupar as *tags* `<concern>` que contém os indícios de ITs de acordo com o interesse ao qual pertencem. O nome desse interesse pode ser recuperado por meio do atributo `name` desta *tag*. Todos os indícios são representados por *tags* `<element>` que possuem os atributos: i) *degree*; ii) *id* e; iii) *type*. O atributo *degree* possui sempre o valor 100 como fixo e não foi encontrada justificativa para a utilização desse atributo. O atributo *type* pode conter dois valores: i) *method* e; ii) *field*, indicando se o indício de IT é um método (*method*) ou um atributo de uma classe (*field*). O atributo id identifica o indício de IT e é o elemento do arquivo CM que possui a maior quantidade de particularidades.

O valor do atributo *id* é o "caminho" do indício dentro do software e segue como padrão: `NomeDoProjeto/src<NomeDoPacote{NomeDoArquivoJava[NomeDaClasse`. Se for um mé-

Tabela 4.1: Sumarização da conversão de tipos do formato Java para o formato CM.

Java	CM
`boolean`	`~Z`
`char`	`~C`
`byte`	`~B`
`short`	`~S`
`int`	`~I`
`long`	`~J`
`float`	`~F`
`double`	`~D`
`NomeClasse`	`~QNomeClasse;`
`Class<? extends boolean[]>`	`~Qclass\<+\[Z>;`
`Class<? extends char[]>`	`~Qclass\<+\[C>;`
`Class<? extends byte[]>`	`~Qclass\<+\[B>;`
`Class<? extends short[]>`	`~Qclass\<+\[S>;`
`Class<? extends int[]>`	`~Qclass\<+\[I>;`
`Class<? extends long[]>`	`~Qclass\<+\[J>;`
`Class<? extends float[]>`	`~Qclass\<+\[F>;`
`Class<? extends double[]>`	`~Qclass\<+\[D>;`
`Class<? extends NomeClasse[]>`	`~Qclass\<+\[QNomeClasse;>;`
`Class<? extends Class>`	`~Qclass\<+QClass;>;`
`boolean[]`	`~\[Z`
`char[]`	`~\[C`
`short[]`	`~\[S`
`int[]`	`~\[I`
`long[]`	`~\[J`
`float[]`	`~\[F`
`double[]`	`~\[D`
`NomeClasse`	`~\[QnomeClasse;`

todo, deve-se adicionar ao padrão `~NomeMétodo` mais os nomes dos tipos de seus parâmetros. Se for um atributo, deve-se adicionar ao padrão `^NomeAtributo`. Se houver classe interna, o padrão para a parte `[NomeDaClasse` é `[NomeDaClasseExterna[NomeDaClasseInterna`. A conversão para o nome de um tipo de parâmetro em formato Java para formato CM estão presentes na Tabela 4.1, onde a primeira coluna mostra como é o nome do tipo em formato Java, e a segunda coluna mostra como esse nome é apresentado de acordo com o formato CM.

O `CMWriter` é utilizado de duas formas: i) dentro do código da SourceMiner (CARNEIRO *et al.*, 2010) para possibilitar a integração do IMM com essa ferramenta e; ii) como um conversor de arquivos IMM, de modo que todo arquivo IMM possa ser convertido para o formato CM, mantendo-se a compatibilidade com versões não integradas da SourceMiner. O detalhamento da implementação de ambas as formas citadas serão detalhadas no próximo capítulo.

4.3.2.3 Classe `CMTypeFormatConverter`

As informações de conversão do formato CM para JAVA foram implementadas na classe `CMTypeFormatConverter` do pacote `helpers`. Essa decisão foi tomada, pois a ferramenta FINT possui um formato para nome de tipos de atributos semelhante à apresentada pela ConcernMapper, sendo somente adicionados alguns caracteres "\" e não sendo utilizado do caractere "∼". Como a remoção desses caracteres é simples e não impacta na conversão do formato CM para Java, essa classe foi criada para agregar essas funcionalidades de conversão e poder ser utilizada na criação dos conversores que recebem como entrada os arquivos CM ou FINT e apresentam como saída arquivos em formato IMM, descritos no próximo Capítulo.

4.3.3 Pacotes `readers` e `readers.model`

O pacote `readers` possui a classe `IMMReader`, responsável por, a partir de um caminho do sistema, ler um arquivo IMM e retornar para o usuário um objeto da classe `MiningTool` com todos os indícios de ITs presentes no arquivo lido. Essa classe também possui uma função chamada `getAllIMMFilesFromPath`, capaz de retornar um objeto do tipo `java.io.File` para cada um dos arquivos IMM presentes em uma determinada pasta.

Essa função auxilia na leitura dos arquivos IMM de um projeto, tendo como parâmetro o caminho de sistema do projeto em análise e retornando todos os arquivos IMM presentes na raiz desse projeto. Para cada um desses arquivos IMM presentes na raiz deve-se chamar a função `readIMM`, para interpretar as informações do arquivo IMM e, com o auxílio da biblioteca XStream, transformar as informações, em formato XML, para o formato de objetos do metamodelo, retornando um objeto da classe `MiningTool`.

O pacote `readers.model` possui a classe `MiningToolConcrete`, utilizada para realizar a leitura de um arquivo que possua um modelo gerado a partir do metamodelo padrão, sem que nenhuma de suas classes tenha sido decorada. Isso facilita na leitura dos arquivos, fazendo com que as ferramentas de visualização, que somente pretendem ler arquivos gerados a partir de modelos sem classes anotadas, possam utilizar as funções de leitura sem implementar uma classe concreta para a classe abstrata `MiningTool`.

4.4 Considerações Finais

Diante das dificuldades encontradas pelos pesquisadores em analisar resultados apresentados pelas diversas ferramentas de MIT, notou-se a necessidade de integrar essas ferramentas

com as de visualização. Além disso, Ceccato *et al.* (2006) evidenciam os benefícios da combinação de diferentes ferramentas de MIT. Carneiro *et al.* (2010), Delfim e Garcia (2013), Bentrad e Meslati (2011) evidenciam a necessidade de exibir as informações referentes aos indícios de ITs por meio de técnicas de visualização.

A fim de possibilitar a exibição dessas informações, é necessário integrar as ferramentas de MIT com as de visualização de software, integração essa que pode ser feita através do metamodelo IMM. Para avaliar a generalidade desse metamodelo e construir um ambiente integrado, algumas ferramentas de MIT e de visualização de software presentes na literatura foram atualizadas para utilizar o formato definido pelo IMM, e são apresentadas no próximo capítulo.

Capítulo 5

UMA UTILIZAÇÃO DO METAMODELO INDICATIONS METAMODEL

5.1 Considerações Iniciais

O metamodelo apresentado no capítulo anterior foi desenvolvido permitindo que ferramentas que apoiam várias técnicas de mineração de interesses transversais (MIT) e de visualização de software possam ser utilizadas em conjunto. Para validar essa premissa do *Indications Metamodel* (IMM), foram integradas quatro ferramentas de MIT e duas ferramentas de visualização de software com considerável utilização conforme relatado na literatura especializada (ABILIO *et al.*, 2012; MCFADDEN; MITROPOULOS, 2012; CHAVEZ *et al.*, 2011; CZIBULA *et al.*, 2011; TREUDE; STOREY, 2010; DAGENAIS; OSSHER, 2007; CARNEIRO *et al.*, 2010; PARREIRA JÚNIOR *et al.*, 2010a).

As ferramentas de MIT consideradas são: i) ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005); ii) ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a); iii) FINT (MARIN *et al.*, 2007) e; iv) XScan (NGUYEN *et al.*, 2011) que utilizam, respectivamente, as técnicas de mineração por análise exploratória, por tipos e texto, por análise de *Fan-in* e por *Concern Peers*. Essas técnicas de MIT e as principais características de cada uma dessas ferramentas foram comentadas no Capítulo 2.

As duas ferramentas de visualização de software que foram escolhidas para o processo de integração são SourceMiner (SILVA A. N.; CARNEIRO, 2012; CARNEIRO *et al.*, 2010) e $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013). As principais características, visões e metáforas utilizadas dessas ferramentas foram apresentadas no Capítulo 3.

Essas ferramentas foram integradas de forma a criar um ambiente no qual possam ser uti-

lizadas em conjunto. Dessa forma, os resultados obtidos pelas ferramentas de MIT podem ser analisados por meio das visões das ferramentas de visualização de software. Essa comparação evidencia os falsos positivos e possibilita a identificação de novos padrões visuais, que evidenciam falsos positivos.

A integração de diferentes ferramentas não é trivial. Existem principalmente dois cenários de integração: i) o código fonte da ferramenta está disponível ou a ferramenta fornece uma interface de integração e; ii) somente um arquivo de formato específico é apresentado como resultado ou forma de importação de dados. O primeiro caso é o ótimo, onde há a certeza de que a integração será realizada, independentemente do esforço empregado para a tarefa. No segundo caso, dependendo do formato do arquivo, pode não ser possível realizar a integração, como em casos onde informações imprescindíveis para integração são omitidos. Em casos onde é possível realizar a integração, o resultado não é um ambiente onde as ferramentas trabalham conjuntamente de forma transparente ao usuário, e sim, um ambiente onde o usuário deve utilizar as ferramentas individualmente de forma a gerar seus arquivos de resultado e convertê-los para o formato esperado pelas demais ferramentas a serem integradas. Das ferramentas escolhidas, as ferramentas ComSCId, XScan, SourceMiner e $SoftVis_{4CA}$ apresentavam o código fonte disponível e as ferramentas ConcernMapper e FINT somente apresentavam um arquivo em formato específico, que precisou ser convertido para o formato do IMM.

A seguir é apresentada a justificativa da escolha de cada uma das ferramentas de MIT e de visualização de software para que fossem integradas por meio da utilização do metamodelo IMM.

A ferramenta ConcernMapper foi escolhida por gerar um arquivo de saída com todos os indícios encontrados, por ser uma ferramenta consolidada e por já estar integrada a SourceMiner (CARNEIRO *et al.*, 2010; TREUDE; STOREY, 2010; DAGENAIS; OSSHER, 2007). Este fato também influenciou a escolha da ferramenta SourceMiner, que além de ser consolidada na área de visualização de software (SCHOTS *et al.*, 2012; ABILIO *et al.*, 2012; CHAVEZ *et al.*, 2011) e já estar integrada com uma ferramenta de MIT (CARNEIRO *et al.*, 2010), possui uma nova versão e arquitetura (SILVA A. N.; CARNEIRO, 2012), o que facilita a inclusão de novas funções para ampliar a sua funcionalidade.

A ferramenta ComSCId foi escolhida por ter sido desenvolvida no grupo de pesquisa ao qual o autor desta dissertação pertence, ter seu código fonte disponível e o pelo conhecimento prévio do autor. A ferramenta XScan foi escolhida por utilizar uma técnica de mineração totalmente automática (NGUYEN *et al.*, 2011); por apresentar bons resultados na identificação de indícios de ITs e também pela disponibilidade, feita pelo autor da ferramenta, de seu código fonte para

que o autor desta dissertação pudesse realizar as alterações necessárias para a integração.

A ferramenta FINT é uma ferramenta consolidada e utilizada como base de comparação com diversas outras ferramentas de mineração (MCFADDEN; MITROPOULOS, 2012; NGUYEN *et al.*, 2011; CZIBULA *et al.*, 2011). Por fim, a ferramenta de visualização $SoftVis_{4CA}$ foi escolhida por ser uma ferramenta de visualização que visa a auxiliar no processo de MIT e na análise do impacto causado por alterações no código fonte, acarretadas pela refatoração dos indícios encontrados, e também por não ser um *plug-in* do Eclipse (Eclipse Foundation, 2013), a fim de demonstrar a generalidade do IMM.

Na Seção 5.2 será detalhado o processo de integração das ferramentas utilizadas, com as alterações realizadas, os problemas superados e como devem ser utilizadas as novas funções inseridas após essa integração. Na Seção 5.3 será apresentada uma prova de conceito que utilizará as quatro ferramentas de MIT e a ferramenta de visualização SourceMiner para identificar os indícios de persistência. Por fim, na Seção 5.4 são apresentadas as considerações finais deste capítulo.

5.2 Procedimentos de Integração

Para que as seis ferramentas mencionadas anteriormente fossem integradas ao IMM foram determinadas atividades que podem ser dividas em dois grupos. O primeiro grupo representa as atividades de atualização do código fonte, e compreende as ferramentas ComSCId, XScan, $SoftVis_{4CA}$ e SourceMiner. O outro grupo representa a criação de conversores, sendo os arquivos de saída, gerados pelas ferramentas ConcernMapper e FINT, lidos e interpretados de modo que as informações pudessem ser convertidas para o formato IMM. Com exceção das alterações realizadas no código fonte da $SoftVis_{4CA}$, todas as outras atividades foram realizadas pelo autor dessa dissertação.

As próximas subseções descrevem detalhadamente as atividades realizadas e os problemas encontrados para que essas seis ferramentas fossem integradas ao IMM. Primeiramente serão descritas as atividades de atualização do código fonte das ferramentas ComSCId e XScan. Logo após serão apresentados os conversores das ferramentas ConcernMapper e FINT. Ao final serão apresentadas as atividades de atualização do código fonte das ferramentas $SoftVis_{4CA}$ e SourceMiner.

Com exceção da $SoftVis_{4CA}$, as ferramentas atualizadas como descrito nas próximas seções e os conversores desenvolvidos, estão disponíveis para download no site:

http://advanse.dc.ufscar.br/integrantes/oscar_tanner

5.2.0.1 ComSCId

A ferramenta ComSCId foi a primeira a ser integrada, visto que seu código fonte estava disponível e considerando o conhecimento prévio deste autor sobre ela. Além de alterações necessárias para a integração com o IMM, foram realizadas algumas atividades de manutenção, que foram necessárias para adequar sua utilização ao sistema operacional Linux, já que a ferramenta estava disponível apenas para o sistema Windows.

O sistema operacional Windows utiliza-se do caractere de contra-barra ("\") para separar os itens de um caminho de sistema, sendo que o caminho de sistema sempre começa com uma letra que representa a unidade de armazenamento onde os arquivos estão salvos. Já o Linux utiliza-se da barra normal ("/"), tanto como caractere de separação dos itens de um caminho de sistema, quanto para representar a unidade principal ou *root* do sistema. Com isso, houve a necessidade de alterar todas as operações que geram arquivos do sistema para que identifiquem o sistema operacional no qual a ferramenta é executada e selecionem corretamente a unidade de armazenamento e os caracteres de separação.

A outra atividade de manutenção foi referente aos arquivos sem extensão, comuns no Linux e considerados como um arquivo de texto qualquer. Já no Windows isso não ocorre, sendo necessário sempre adicionar a extensão ".txt" em arquivos texto. Com isso ao se executar a ComSCId em um projeto que tivesse algum arquivo sem extensão, o mesmo não o interpretava, gerando uma exceção que interrompia a sua execução. Fez-se necessário localizar, no código fonte da ferramenta, onde a leitura de arquivos sem extensão gerava erro e criar um novo laço condicional para que os mesmos fossem ignorados. Uma vez que por não terem extensão, não eram arquivos do código fonte e, portanto, não eram importantes para a execução do sistema.

Outro problema encontrado, que não foi identificado logo no início e mascarou as correções citadas acima, foi o de que a versão 3.7.2 ou Indigo do Eclipse não apresenta em suas bibliotecas de sistema uma classe para a atualização das abas da ComSCId chamada `SaveAllActions` do pacote `org.eclipse.ui.internal`. Essa versão do Eclipse estava sendo utilizada por este autor, mas não era usada pelos desenvolvedores anteriores (PARREIRA JÚNIOR *et al.*, 2010a). Como essa classe está presente nas versões anteriores e posteriores a versão Indigo, decidiu-se adicionar um novo requisito de sistema, que determina que essa ferramenta não deve ser executada na versão Indigo do Eclipse.

Finalizadas as alterações mencionadas, foi iniciado o processo de integração. A ComSCId possui uma estrutura de objetos semelhante a do IMM, que utiliza um metamodelo similar como base. Essa estrutura possibilitou que as informações dos indícios de ITs encontrados pela

ComSCId fossem convertidas diretamente por estruturas semelhantes do IMM. Essa conversão utilizou-se da extensão das classes do IMM, que receberam um novo construtor capaz de converter um objeto de uma classe da ComSCId em um objeto do IMM. Na Figura 5.1 é exibido o código fonte da classe `Package` do IMM, decorada para exemplificar essa situação. Essa classe possui um construtor que recebe como parâmetro uma classe `Package` da ComSCId. Para cada uma das classes desse objeto é criado um novo objeto da classe `Class` do IMM, que recebe como parâmetro o objeto no formato da ComSCId, ficando sob a responsabilidade desse novo construtor realizar as demais atividades de conversão. Todas as estruturas a serem convertidas da ComSCId são homônimas às do IMM, com exceção da classe `Method` que na ComSCId é chamada de `Operation`.

```
package br.ufscar.dc.imm.model;

import java.util.Iterator;

public class Package extends br.ufscar.dc.metamodel.model.Package {

    public Package( br.ufscar.dc.dmasp.model.Package aPackage ) {
        super( aPackage.getName() );

        Iterator<br.ufscar.dc.dmasp.model.Class> it = aPackage.getClasses().iterator();
        while( it.hasNext() ) {

            br.ufscar.dc.dmasp.model.Class next = it.next();

            // Creating Class from ComSCId class
            if( next.hasConcern() ) {
                addClass( new Class( next ) );
            }
        }
    }

}
```

Figura 5.1: Classe Package decorada.

Esse tipo de conversão é interessante por utilizar os principais pontos positivos da OO, dividindo o trabalho de conversão das estruturas da ComSCId entre as do IMM, o que permite que o código fonte gerado seja de fácil manutenção. Outro ponto a destacar é que esse processo somente apresenta erros de omissão por parte do engenheiro de software e de compilação. Sendo os erros de compilação de fácil identificação, já que o próprio ambiente de desenvolvimento Eclipse os identifica.

Após decorar todas as classes, para que a integração funcionasse corretamente, foi necessário inserir um novo atributo na classe `Parameter` da ComSCId chamado `typeComplete`. Esse atributo armazena exatamente como é apresentado o tipo do atributo no código fonte Java, permitindo que esse valor seja posteriormente informado para as ferramentas de visualização de software. Como exemplo, considere um parâmetro cujo tipo seja `Class<? extends AnotherClass>`, um parâmetro que aceita qualquer objeto que estenda `AnotherClasse`. Ao

armazenar essa informação em suas estruturas padrão, a ComSCId armazena somente a informação `Class` como tipo desse parâmetro, omitindo os demais caracteres necessários para uma correta identificação desse método pelas ferramentas de visualização de software. Apesar da inclusão do atributo `typeComplete`, a ComSCId ainda não consegue identificar corretamente atributos que são vetores e apresentam o caractere de abre e fecha colchetes ao lado do nome do atributo, ex: "`String args[]`". Neste exemplo a ComSCId atribui somente o valor "`String`" para o atributo `typeComplete`. Uma vez finalizada a integração foi adicionado um menu para possibilitar a criação de um arquivo IMM sempre que o engenheiro de software desejar. Para isso uma nova entrada, chamada "*Generate Indications Metamodel (IMM)*", foi inserida no menu *Reverse Engineering* (Figura 5.2), já existente nessa ferramenta.

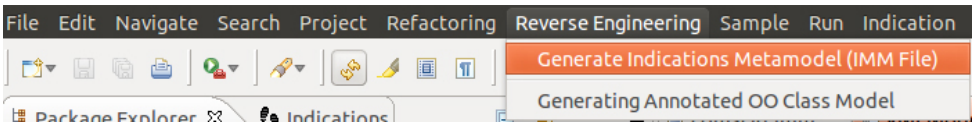


Figura 5.2: Menu para criação do arquivo IMM com os indícios de ITs identificados pela ComSCId.

Ao selecionar esse menu, uma função que instancia um objeto da classe `MiningTool` do IMM é chamada e com a utilização de seu construtor decorado, é iniciado o processo de conversão das classes da ComSCId para as classes do IMM. Esse construtor também recebe como parâmetro identificações da ComSCId a serem armazenadas em seus atributos, como nome da ferramenta e tipo da técnica de mineração, respectivamente "ComSCId" e "Types and Text". Após a execução desse construtor, o objeto da classe `MiningTool` apresenta todas as informações referentes aos indícios de ITs identificados pela ComSCId. Para finalizar o processo é, então, chamado o escritor de IMMs disponibilizado pelo *plug-in* do IMM que cria um arquivo, chamado "comscid.imm", na raiz do projeto analisado. Esse arquivo contém todos os indícios identificados pela ComSCId e pode ser utilizado pelas ferramentas de visualização para exibi-los.

5.2.0.2 XScan

A ferramenta XScan também necessitou de algumas alterações para que pudesse ser integrada com as demais ferramentas. Assim como a ComSCId, essa ferramenta também apresentou problemas com os caminhos de sistema, que estavam todos configurados somente para o Windows. Assim houve a necessidade de atualizar todas as operações que geram caminhos de sistema para também apoiarem a utilização do Linux. Outra alteração necessária foi a identificação dinâmica do projeto analisado. A versão original (NGUYEN *et al.*, 2011) necessitava

que o caminho de sistema e o do projeto a ser analisado fossem informados dentro de seu código fonte, necessitando que o *plug-in* fosse recompilado sempre que se desejasse analisar um novo projeto. Outro caminho, que necessitava ser identificado dinamicamente, era o da pasta Output, que sempre era salva na raiz do *workspace* do Eclipse. Para possibilitar a integração houve a alteração desse caminho para que a pasta *Output* fosse gerada dentro da raiz do projeto analisado.

Para a identificação dinâmica do projeto analisado, alterou-se sua interface de execução de um menu principal (Figura 5.3) para um item de menu que aparece ao clicar com o botão direito do mouse sobre um projeto (Figura 5.4). Esse novo item de menu é capaz de identificar o projeto e iniciar as atividades do processo de mineração por AspectWiz e por CBFA, que foram apresentadas no Capítulo 2. Esse menu também possibilita que os resultados obtidos sejam armazenados pela pasta *Output* na raiz do projeto analisado. Finalizadas as atividades de mineração, iniciaram-se as atividades de conversão para o formato IMM. Como não foi possível identificar o formato de objetos utilizados pela ferramenta XScan, optou-se por converter as informações no momento em que seriam gravadas no arquivo de saída da pasta *Output* e considerando o formato utilizado nesse arquivo.

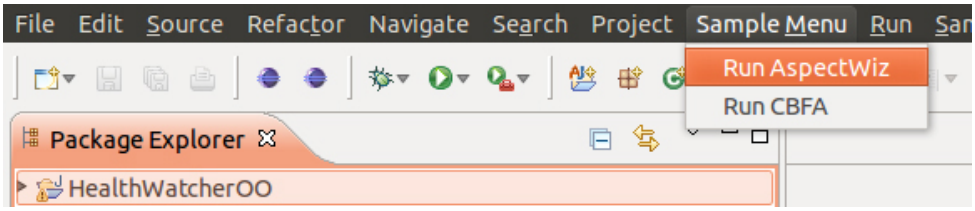


Figura 5.3: XScan, execução iniciada por menu principal.

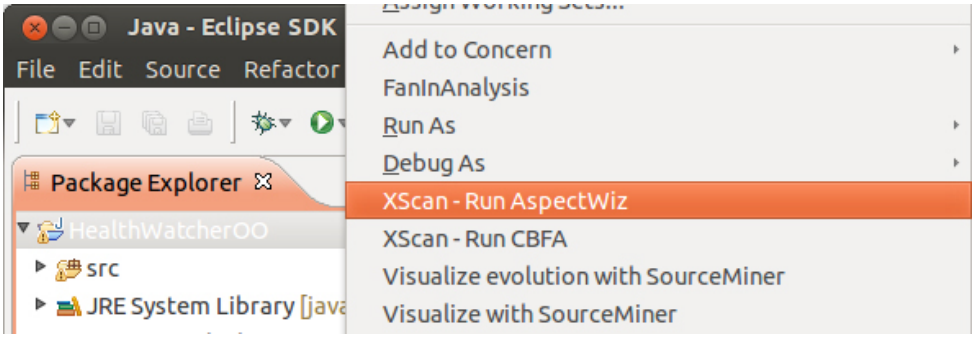


Figura 5.4: XScan, execução por item de menu acessado ao clicar com botão direito do mouse sobre o projeto a ser analisado.

Para entender o formato do arquivo gerado, é preciso saber que a técnica de mineração por *Concern Peers*, implementada pela XScan, fornece os resultados por meio de grupos de métodos que contém contextos similares. Com isso optou-se por fazer com que cada um dos grupos identificados fosse considerado um IT, sendo que os métodos agrupados seriam os seus

indícios. Assim, ao utilizar a integração, cabe ao engenheiro de software analisar o arquivo de saída da XScan, identificar os grupos que deseja visualizar e selecioná-los na ferramenta de visualização para que sejam exibidos.

Já as informações contidas nos arquivos de saída da XScan, referentes aos métodos agrupados, seguem um padrão próprio que, de forma geral, determina que o caminho de projeto de todos os indícios identificados deve começar com a letra maiúscula "L". Depois ocorre a separação do nome do pacote, sub-pacotes e classe com o caractere "/" ao invés do caractere padrão do Eclipse que é o ponto final. O nome do método deve ser separado do nome da classe pelos caracteres ";.", sendo que os parâmetros desse método são exibidos dentro de parênteses logo após seu nome. Se os parâmetros possuírem um tipo primitivo, a conversão deve ser realizada de acordo com as regras de conversão implementadas dentro da classe `Type` do metamodelo IMM, por meio da função estática `returnPrimitiveTypeName`, apresentada no Capítulo 4. Essa função converte o conjunto de caracteres utilizados na linguagem Java para um único caractere maiúsculo, por exemplo: o tipo primitivo int é convertido para a letra "I".

Quando os tipos dos parâmetros não são primitivos, esse padrão determina que sejam incluídas todas as informações, referentes aos pacotes e sub-pacotes, aos quais essa classe pertence. Os nomes dos pacotes, sub-pacotes e classe devem ser separados pelo caractere "/" e finalizados com o caractere ";". Esse último caractere é responsável por separar os parâmetros com tipos não primitivos, porém tipos primitivos, representados por somente um único caractere, não apresentam nenhum caractere de separação.

As informações desse arquivo foram convertidas para o formato IMM, permitindo a geração de um arquivo na raiz do projeto analisado, de forma que essas informações possam ser analisadas pelas ferramentas de visualização integradas. Para tanto é gerado um arquivo para cada tipo de mineração da XScan, ou seja, são gerados os arquivos "xscan_aspect_wiz.imm" e "xscan_cbfa.imm", que podem ser vistos na Figura 5.5.

5.2.0.3 ConcernMapper

A ferramenta ConcernMapper foi a que mais exigiu esforços para a integração, visto que o seu arquivo de saída possui um formato próprio com muitas particularidades e pouca documentação (ROBILLARD, 2013; ROBILLARD; WEIGAND-WARR, 2005). Como o esforço para entendimento desse formato foi considerável, tomou-se a decisão de criar também um escritor para esse formato. O conhecimento obtido contribuiu para manter a compatibilidade com as versões não integradas da SourceMiner.

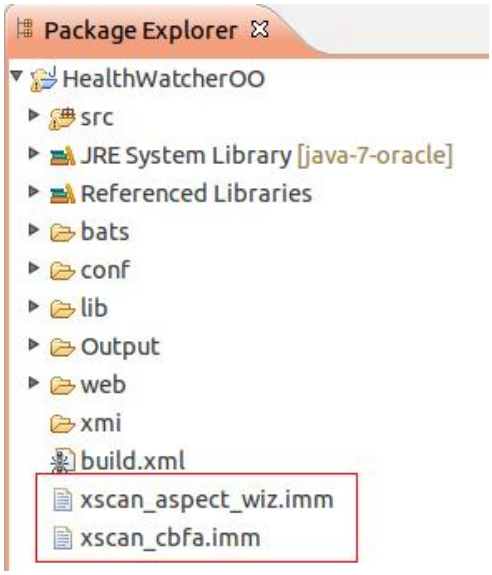


Figura 5.5: Arquivos IMM gerados pela XScan na raiz do projeto analisado.

Esse formato, já comentado no Capítulo 4, possui como pontos principais o formato de um arquivo XML que não deve conter nenhum caractere de quebra de linha e deve conter as *tags* `<model>`, `<concern>` e `<element>`. A primeira *tag* representa a raiz do XML, a segunda agrupa todos os indícios de um determinado IT e a última representa o indício encontrado, ou sementes como foram denominadas pelo autor da ConcernMapper.

Como não havia o código fonte disponível da ConcernMapper, para a sua integração foi necessário criar um conversor capaz de entender o arquivo de saída dessa ferramenta e gerar um arquivo no formato IMM. Foi criado então um item de menu, que aparece quando se clica com o botão direito do mouse sobre qualquer arquivo com a extensão ".cm", com a chamada de "*Transform from CM file to IMM file*" (Figura 5.6). Esse conversor lê o arquivo CM, identifica todos os ITs e indícios representados, converte-os para o formato do IMM e gera um arquivo na raiz do projeto denominado de "concernmapper.imm", como pode ser visto na Figura 5.7.

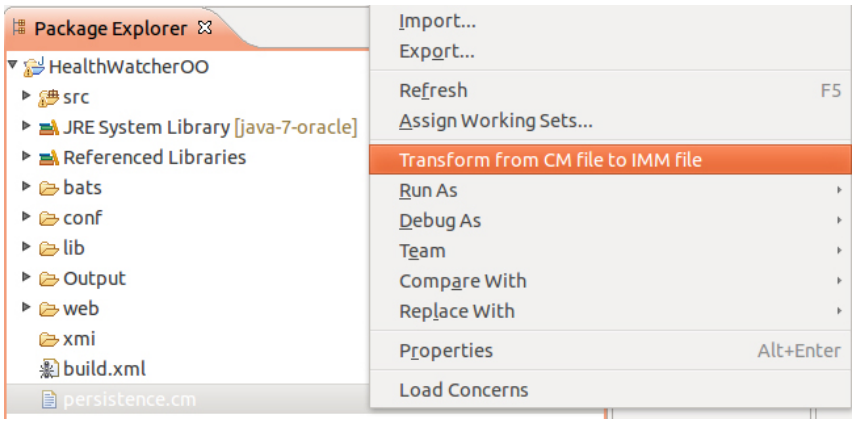


Figura 5.6: Menu para converter um arquivo CM para o formato IMM.

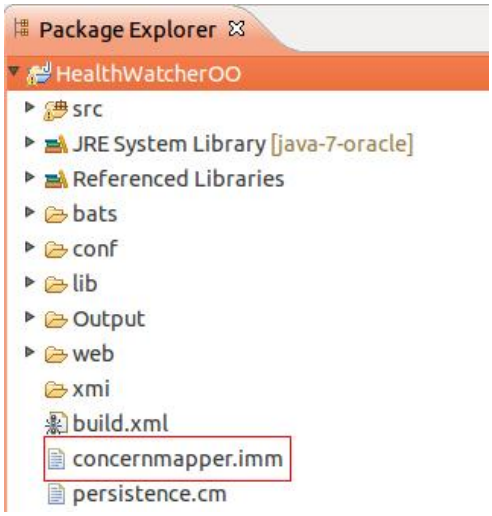


Figura 5.7: Arquivo IMM criado a partir do arquivo CM.

5.2.0.4 FINT

A ferramenta FINT também não possui seu código fonte disponível. Para sua integração também foi necessária a criação de um conversor, capaz de entender os indícios representados por seu arquivo de saída, de modo a convertê-los para o formato IMM. A conversão é realizada quando houver um clique com o botão direito do mouse sobre qualquer arquivo com a extensão ".txt" e for selecionado o item de menu "Transform from FINT file to IMM File" (Figura 5.8). A ferramenta FINT não possui uma extensão padrão para seus arquivos de saída, como a ConcernMapper. Assim, foi necessário configurar esse menu para que fosse exibido para qualquer arquivo texto presente em um projeto. O engenheiro de software tem a responsabilidade de solicitar a conversão somente dos arquivos que realmente foram gerados pela FINT.

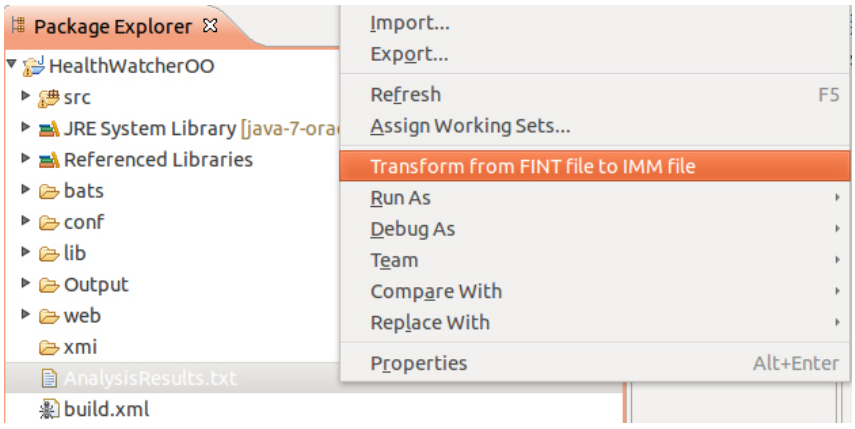


Figura 5.8: Item de menu para converter o arquivo da ferramenta FINT para o formato IMM.

Essa ferramenta, por utilizar a técnica de MIT por análise de *Fan-in*, gera o arquivo de saída em formato análogo aos grupos da XScan, sendo os indícios agrupados de acordo com o método

chamado. A partir disso, no processo de conversão, todo método chamado é considerado um IT e os métodos chamadores são os indícios desse IT.

Para facilitar a análise dos indícios nas ferramentas de visualização foi criado um IT chamado "*All FINT indications in one concern*", que deve ser utilizado somente quando a análise resultante exibir métodos de somente um IT, pois agrupa todos os métodos chamados que foram identificados.

A primeira linha do arquivo gerado pela FINT contém a quantidade de métodos chamados que foram identificados. Todo método chamado é representado diretamente por seu caminho no projeto. Esse caminho, para a representação dos pacotes, sub-pacotes, nome da classe e nome do método, segue o padrão Java, ou seja, esses nomes devem estar separados pelo caractere de ponto final. Esse formato não apresenta nenhum tipo de separação entre o nome da classe e o nome do método, a não ser o caractere de ponto final. Portanto, para reconhecer qual é o nome da classe optou-se por procurar o primeiro nome cuja primeira letra é maiúscula, uma vez que em Java, uma boa prática de programação, é não ter pacotes que comecem com letra maiúscula. Esse fato leva a uma limitação dessa integração: se existir algum pacote cujo nome comece com letra maiúscula, que é permitido pela linguagem Java, ocorrerá um erro na conversão e a integração na funcionará.

Os parâmetros de um método são representados entre parênteses logo após o nome do método e possuem um formato específico da FINT, semelhante ao da ConcernMapper apresentado no Capítulo 4. As principais diferenças são que a FINT utiliza alguns caracteres a mais de "\" para representar alguns tipos, e não utiliza o caractere "∼". Finalizada a identificação de um método é utilizado o caractere de dois pontos ":" para separar o método de seu valor segundo a métrica *Fan-in*. Os métodos chamadores ou indícios, como considerados para a integração, são exibidos logo após o método chamado e seguem as mesmas regras do método chamado. Porém a identificação de um método chamador inicia-se sempre com dois caracteres de espaço, um caractere de maior ">" e mais um caractere de espaço. A separação, entre os métodos chamadores é realizada por meio do caractere de quebra de linha.

5.2.0.5 $SoftVis_{4CA}$

No intuito de possibilitar que a $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013) interprete os arquivos IMM e use as informações contidas neles, foram necessárias modificações em seu código fonte, realizadas pelos autores da ferramenta. Essas modificações compreenderam a criação de um pacote chamado *ConcernsView*, que é responsável por agrupar as classes responsáveis pela obtenção e visualização dos indícios de ITs. Essas classes realizam, de ma-

neira geral, duas atividades:

- **Importação dos interesses:** os arquivos IMM presentes em um determinado diretório são interpretados de modo que os indícios de ITs são obtidos, possibilitando que os métodos afetados sejam coloridos;
- **Criação de uma lista de IT:** é criada uma tabela com todos os ITs que afetam o sistema, permitindo modificar a cor atribuída aleatoriamente para diferenciá-los e selecionar ITs para a coloração de seus métodos afetados.

Um questionário foi elaborado para a obtenção do *feedback* dos autores da $SoftVis_{4CA}$ quanto à realização da integração. Foi relatado pelos mesmos que não houve dificuldade para a utilização do metamodelo, sendo necessário, aproximadamente, três horas para a realização da integração, que compreendeu as atividades de entendimento do metamodelo e atualização do código fonte.

O resultado da integração pode ser visto na Figura 5.9. Na área A é apresentada a tabela contendo todos os ITs identificados, no caso o IT de persistência identificado por duas ferramentas de MIT, ComSCId e ConcernMapper, respectivamente representados pelas cores azul e vermelho. Na área B é apresentada a visão Inter-Métodos, explicada no Capítulo 2. Nessa visão são apresentados os indícios identificados, bem como os um método em destaque e seus "vizinhos" identificados pela cor rosa.

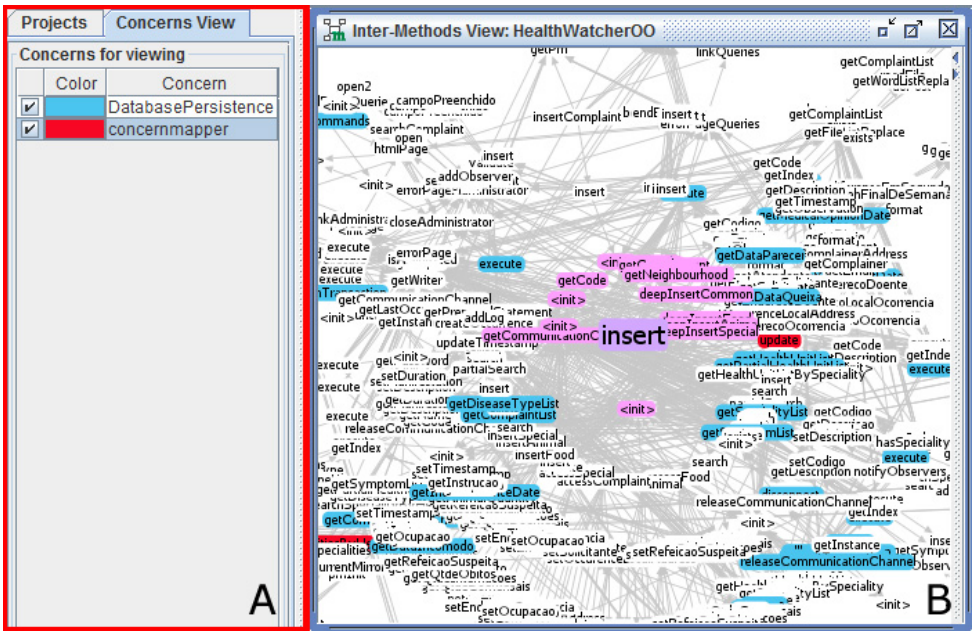


Figura 5.9: Exemplo da integração com a ferramenta $SoftVis_{4CA}$.

5.2.0.6 SourceMiner

A ferramenta SourceMiner, por possuir duas versões necessitou de duas abordagens de integração e chamadas daqui em diante de versão 1 (CARNEIRO *et al.*, 2010) e versão 2 (SILVA A. N.; CARNEIRO, 2012).

O código fonte da versão 1 foi disponibilizado para o autor desta dissertação e, portanto, foi atualizado para possibilitar a leitura dos arquivos IMM. Para evitar um extenso processo de atualização no código fonte da SourceMiner, decidiu-se manter a leitura dos arquivos CM, utilizando o escritor de CM presente no *plug-in* do metamodelo para converter os arquivos IMM. Os arquivos CM gerados por meio dos IMM, são lidos pela SourceMiner e depois apagados, tornando essa conversão opaca para o usuário.

A versão 2 é divida em três camadas: uma de Modelagem, uma de Controle e outra de Visualização. A camada de modelagem é responsável pela importação dos dados que serão exibidos pelas visualizações; a camada de controle é responsável por processar os dados importados e; a camada de visualização é responsável pela definição das visualizações que irão exibir os dados importados. Um novo módulo foi criado, a partir da estrutura existente, e é capaz de ler os arquivos IMM, analisar os indícios presentes e informar a SourceMiner quais são as estruturas afetadas. Dessa forma a versão 2 é capaz de ler um arquivo IMM, não necessitando de nenhum tipo de conversão. Essa versão apresenta como limitação a não identificação de métodos sobrecarregados e a não aceitação de nomes de ITs com caracteres especiais.

Todas as versões da SourceMiner são integradas com a ferramenta ConcernMapper. Para manter a compatibilidade, decidiu-se criar um conversor para que, qualquer arquivo IMM presente na raiz do projeto, pudesse ser transformado em um arquivo CM. Assim, mesmo as versões da SourceMiner que não foram atualizadas para reconhecerem os arquivos IMM, podem ser utilizadas para compor um ambiente integrado com as ferramentas de MIT. Essa conversão é realizada quando houver um clique com o botão direito do mouse sobre um arquivo com a extensão ".imm" e for selecionado o menu "*Transform from IMM File to CM File*", como apresentado na Figura 5.10, sendo o arquivo CM gerado na raiz do projeto.

Somente após a realização de testes de utilização das ferramentas de MIT com as duas versões da SourceMiner, percebeu-se a necessidade de algumas alterações, que não estavam evidentes antes da integração e que somente foram realizadas na versão 1 por seu código fonte estar disponível. A primeira alteração foi realizada quanto à aba "*ConcernFilterView*", que exibe os ITs existentes no código fonte e permite a seleção das cores com as quais esses IT serão pintados. Ao utilizar ferramentas como a FINT e a XScan, que por terem seus grupos

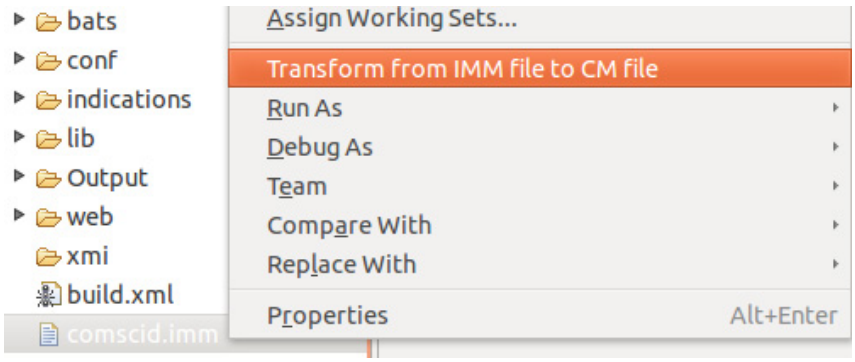


Figura 5.10: Item de menu para converter qualquer arquivo IMM em um arquivo CM.

e métodos chamados tratados como IT, havia problemas na exibição deles. Os ITs no final da lista não eram exibidos pela falta de *scrollbar* vertical e os IT que tinham nomes com grande quantidade de caracteres (nomes grandes) não eram completamente exibidos por falta de *scrollbar* horizontal. Para colorir os indícios de IT, por meio da "*ConcernFilterView*", é necessário selecionar a cor e clicar em um *checkbox* localizado após o nome do IT. Como comentado, pela falta do *scrollbar* horizontal havia problemas de exibição e coloração dos mesmos, pois o *checkbox* não era exibido. Outra alteração realização foi colocar o *checkbox* antes do nome do IT, permitindo que ela fosse selecionada sem necessidade da utilização da *scrollbar*.

A visualização de *Treemap* quando utilizada com os indícios coloridos notou-se necessidade de alteração, pois não havia um *link* para o código fonte de modo que se visualizasse diretamente da *Treemap* o método afetado, facilitando assim a análise do código fonte. Esse *link* foi inserido no último nível de zoom dessa visualização. Assim a partir dessa alteração o engenheiro de software, quando na *Treemap* com zoom em nível de classe, ao clicar em um retângulo que representa um método, tem a possibilidade de ver o código fonte da classe na linha onde o método é declarado.

Outra necessidade foi a de uma perspectiva, que agrupasse todas a abas da SourceMiner. Para utilizar essa ferramenta é necessário selecionar uma a uma todas as suas abas e organizá-las de forma que possam ser corretamente visualizadas. Como as ferramentas de mineração também possuem suas abas e algumas suas próprias perspectivas, ao se alternar entre as abas para realizar o processo de mineração integrado à visualização, algumas das abas da SourceMiner eram fechadas e necessitavam serem abertas novamente. Para resolver essa situação e melhorar a usabilidade da ferramenta foram criadas duas perspectivas, denominadas: SourceMiner e SourceMiner (FINT), respectivamente apresentadas nas Figuras 5.11 e 5.12. A diferença entre essas perspectivas é o posicionamento da aba "*ConcernFilterView*", que ao exibir ITs da ferramenta FINT, necessita de um maior espaço horizontal por causa dos nomes dos ITs.

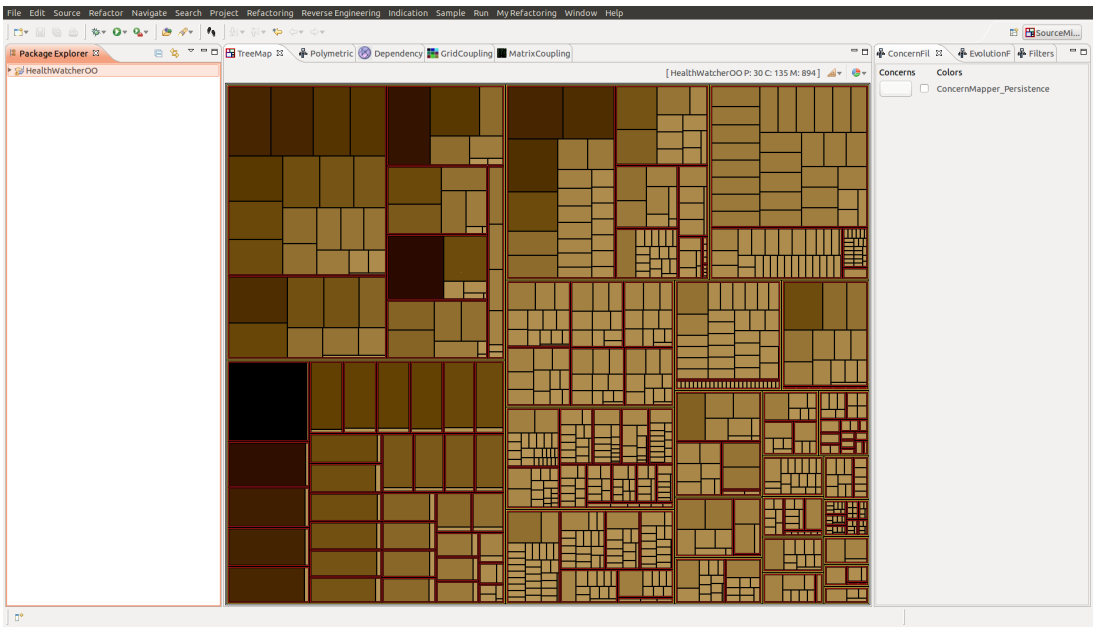


Figura 5.11: Perspectiva SourceMiner.

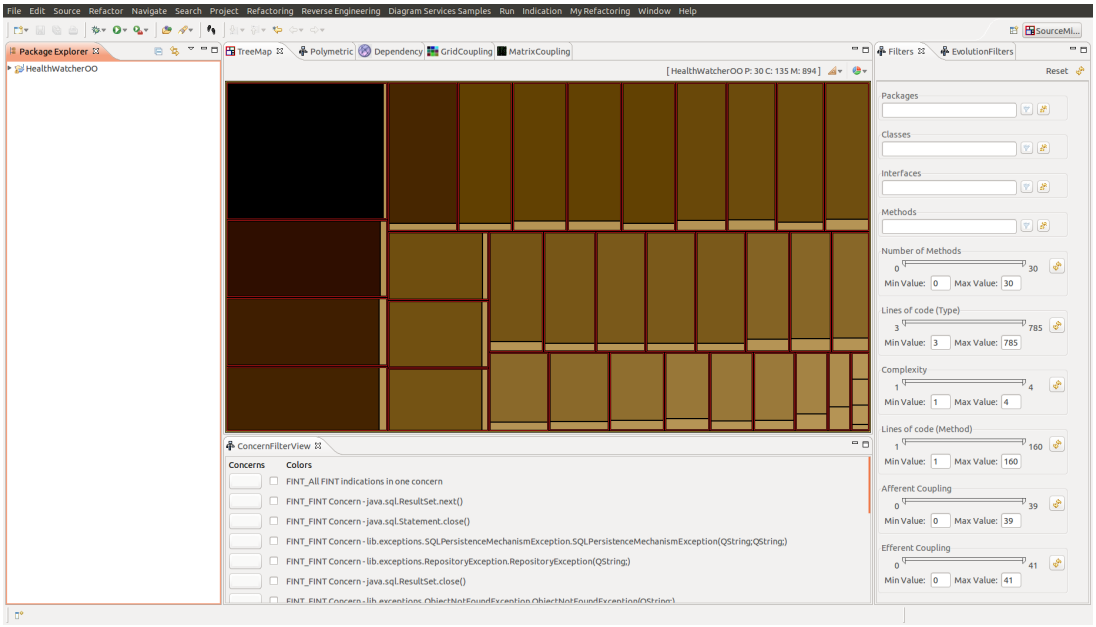


Figura 5.12: Perspectiva SourceMiner (FINT).

5.3 Prova de Conceito

Na seção anterior foram comentadas as atualizações realizadas em cada uma das seis ferramentas utilizadas neste projeto. Para verificar a validade da integração será descrita uma prova de conceito realizada, na qual as quatro ferramentas de MIT e a SourceMiner, versão 1 (CARNEIRO *et al.*, 2010), serão utilizadas. O sistema GrenJ (DURELLI, 2008), um *framework* para geração de aplicações na área de gestão de negócios, desenvolvido no mesmo grupo de pesquisa deste autor, será utilizado para identificar o indício de persistência em banco de dados (que a

partir desse momento será referenciado somente como persistência) em seu código fonte. A versão utilizada do GrenJ foi adaptada para a realização dessa prova de conceito, de forma que algumas de suas classes foram removidas para evitar que o processo de mineração ficasse muito extenso considerando que esse *framework* tem aproximadamente 70 classes e 18 mil linhas de código.

A escolha para a utilização da SourceMiner, versão 1, decorre de maior experiência do autor com essa versão da ferramenta e também por causa das alterações realizadas como comentadas anteriormente. Todas as visões dessa ferramenta podem e devem ser utilizadas em um processo de MIT e visualização, porém visando a simplicidade e facilidade de entendimento dessa prova de conceito, a visão *Treemap* foi escolhida como uma visão geral de referência, sendo utilizada nos exemplos apresentados.

Para a prova de conceito decidiu-se utilizar o IT de persistência. Essa escolha foi realizada por três motivos: i) é um IT bastante comum, onde grande parte dos programadores possui conhecimento sobre ele; ii) é possível encontrar uma infinidade de sistemas afetados por ele, para serem utilizados como exemplo e; iii) a ferramenta ComSCId apresenta um conjunto de regras pré-cadastradas para identificá-lo, facilitando o entendimento da Prova de Conceito e deixando-a mais breve. Vale lembrar que, apesar da prova de conceito utilizar somente esse IT, a abordagem apresentada pode ser utilizada para a identificação qualquer IT.

A ferramenta ComSCId será utilizada para iniciar o processo de mineração por possuir um conjunto de regras para o IT de persistência, que dispensam o conhecimento inicial do sistema. Os indícios encontrados na primeira execução da ComSCId podem ser vistos na Figura 5.13. Com a coloração dos métodos pode-se encontrar quatro padrões diferentes:

1. Um pacote (área A) com vários indícios do IT de persistência, no qual quase todas as suas classes possuem ao menos um método afetado. Provavelmente, esse pacote é o responsável pela implementação da persistência nesse sistema, o que pode ser confirmado por seu nome, `grenj.persistence`;

2. Ainda referente ao pacote do item 1., no qual somente uma de suas classes possui um método não afetado pelo IT de persistência (área A1). Por ser a única classe do pacote de persistência não afetada, há maior probabilidade de que se trate de um falso negativo. Portanto, necessita de uma análise mais detalhada que será realizada posteriormente.

3. Na área B há somente uma classe do pacote `grenj.util` que possui métodos afetados pelo IT de persistência. Existem duas possibilidades de interpretação para esse padrão: a) ele pode representar falsos positivos; ou b) ele pode representar erros de modularização,

visto que, se realmente forem indícios do IT de persistência, essa classe deveria estar implementada no pacote `grenj.persistence`. Nos dois casos deve ser realizada uma análise mais detalhada, o que será feito posteriormente.

4. O quarto padrão indica a presença de alguns indícios espalhados pelas classes do pacote `grenj.model` (área C). Esse padrão pode indicar falsos positivos, que serão analisados logo em seguida.

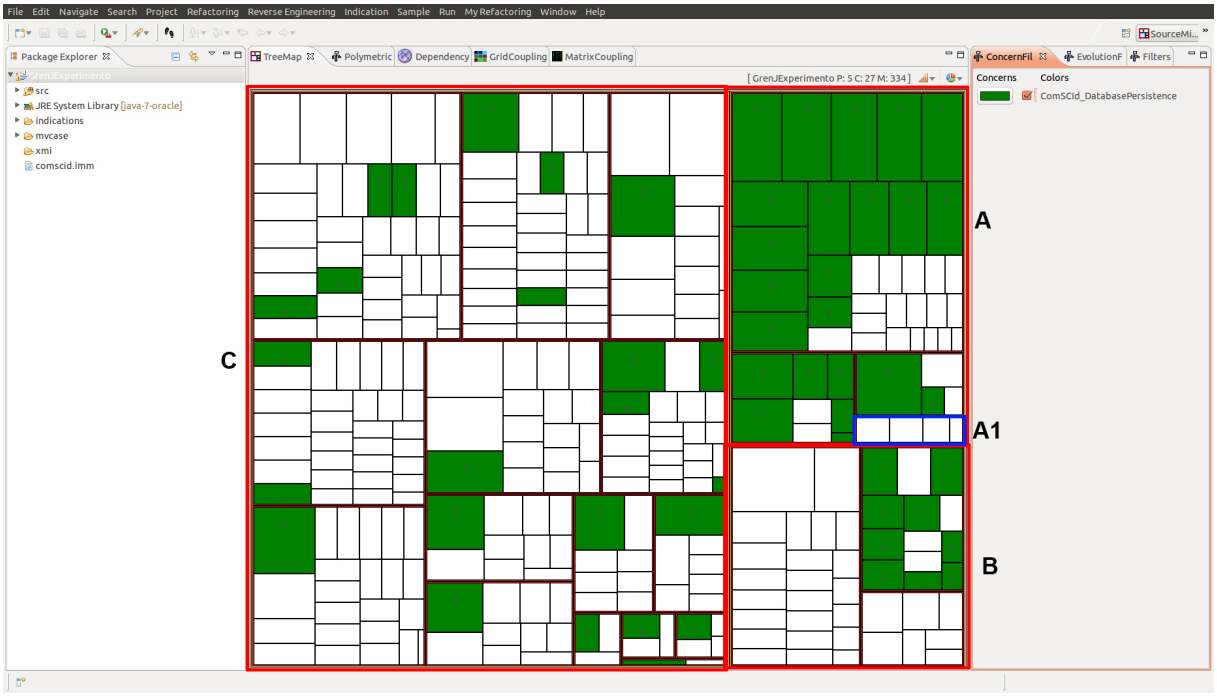
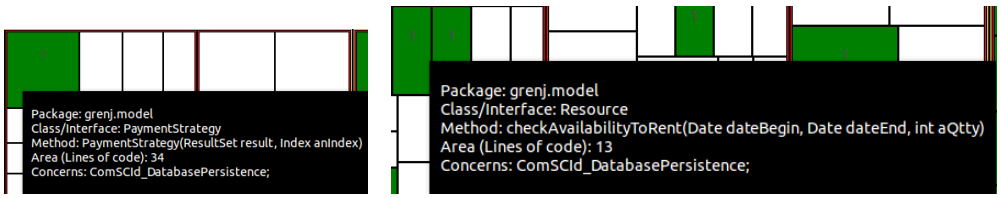


Figura 5.13: Indícios encontrados na primeira execução da ferramenta ComSCId.

Os indícios existentes na área C são formados por dois tipos de métodos, sendo que um representante de cada um desses tipos de métodos foi selecionado como exemplo e podem ser visualizados na Figura 5.14. O primeiro tipo, apresentado na Figura 5.14(a), refere-se a métodos que recebem como parâmetro um `ResultSet` do pacote `java.sql`. Após a análise do código fonte percebe-se que eles realmente são indícios do IT de persistência. O segundo tipo, apresentado na Figura 5.14(b), é composto por métodos que recebem como parâmetro um ou mais objetos do tipo `Date`, que pertencem ao pacote `java.sql`. Após a análise do código fonte percebe-se que os parâmetros do tipo `Date` somente representam datas normais e foram escolhidos por preferência do desenvolvedor em relação à classe Date do pacote `java.util` e portanto são falsos positivos. Para eliminar esses falsos positivos deve-se alterar as regras do ComSCId para que objetos do tipo `java.sql.Date` não sejam mais considerados indícios do IT de persistência. Os indícios encontrados, após essa atualização de regras e nova execução da ferramenta, podem ser vistos na Figura 5.15.



(a) Primeiro tipo (b) Segundo tipo

Figura 5.14: Exemplos de métodos presentes na área C da Figura 5.13

Figura 5.15: Indícios encontrados após remover a classe java.sql.Date das regras da ComSCId.

Ao analisar novamente os padrões apresentados pela coloração dos indícios de IT de persistência (Figura 5.15), foi notado que a retirada da classe `java.sql.Date`, das regras da ComSCId, não interferiu nos indícios apresentados na área A. Na área C, restaram somente os métodos que recebem como parâmetro um `java.sql.ResultSet`, e que realmente são indícios do IT de persistência. Já na área B, quase todos os indícios desapareceram, confirmando que eles eram falsos positivos. Ainda existe um único método afetado, chamado `getToday()` (Figura 5.16), que necessita de melhor análise. Esse método retorna a data atual por meio de uma variável java.sql.Date o que não é um indício do IT de persistência. Sua identificação pelo ComSCId ocorre, pois possui uma variável chamada `sqlDate`, que é identificada por ter as letras "sql" em seu nome.

Para resolver esse problema pode-se alterar o código fonte, trocando o nome dessa variável ou removendo a regra do ComSCId. Se o código fonte for alterado, o problema será resolvido para esse método, porém se existirem outros falsos positivos como esse, eles continuarão a ser

```java
/**
 * Returns the today's date.
 * @return an instance of <code>java.sql.Date</code> representing the
 * today's date.
 */
public static java.sql.Date getToday() {

    java.util.Date Datetoday = new java.util.Date();
    //format definition
    java.text.SimpleDateFormat sdf = new java.text.SimpleDateFormat( "yyyy-MM-dd" );
    //creating SQL Date
    java.sql.Date sqlDate = java.sql.Date.valueOf( sdf.format( Datetoday ));

    return sqlDate;
}
```

Figura 5.16: Método `getToday()`.

identificados. Por outro lado, se a regra for removida, deve-se analisar se nenhum outro indício, previamente identificado e que realmente é afetado pelo IT de persistência, passou a ser um falso negativo. Neste trabalho, optou-se por remover a regra e utilizar a visualização proporcionada pela SourceMiner para validar se nenhum indício deixou de ser identificado. Assim, o novo conjunto de indícios pode ser visto na Figura 5.17, sendo que não existem mais indícios no pacote `grenj.util` e não foi observada alteração nos demais indícios apresentados.

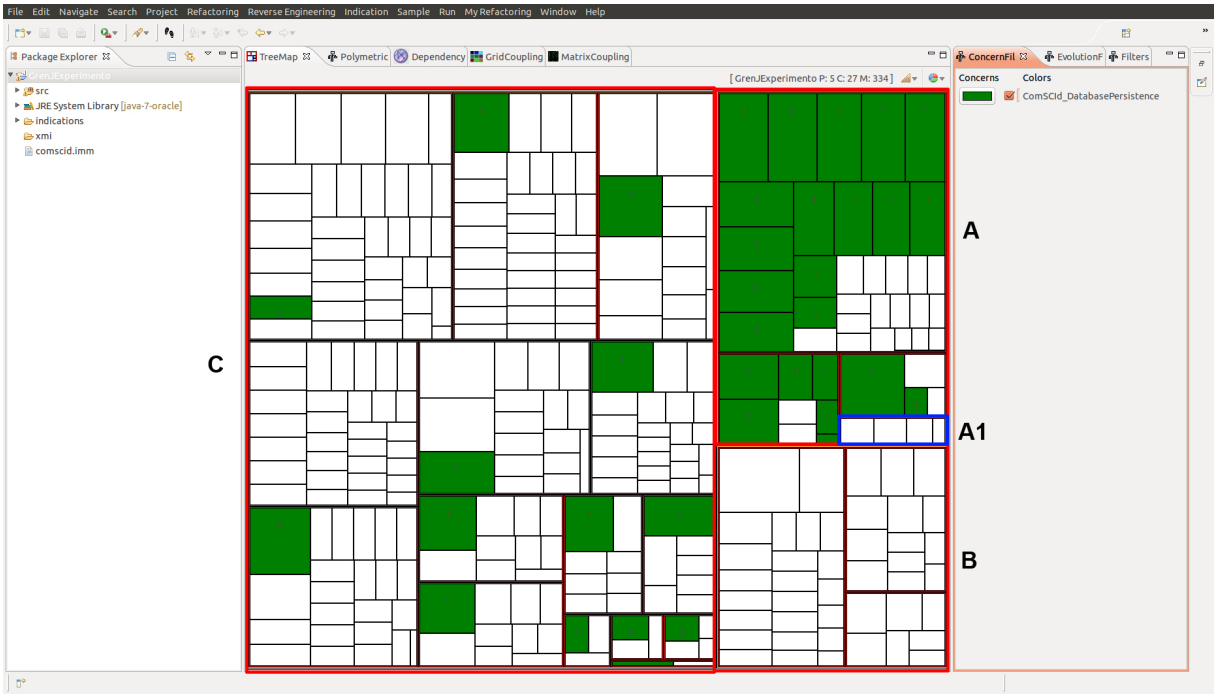


Figura 5.17: Indícios identificados pela ComSCId após a remoção da regra que identificava a função getToday().

A classe `RowNotFoundException`, da área A1, deve ser analisada por ser a única classe do pacote `grenj.persistence` que não possui nenhum método afetado pelo IT de persistência. Na Figura 5.18 pode ser visto o cabeçalho dessa classe, que pelos comentários do seu autor, indica que tem como responsabilidade representar uma exceção ocorrida quando um determi-

nado registro não for encontrado em uma tabela do banco de dados. Assim, essa classe é um indício do IT de persistência e deve ser incluída nas regras do ComSCId. Os indícios encontrados por esse novo conjunto de regras não apresenta alteração quanto aos indícios apresentados pela Figura 5.17, sendo que mesmo os métodos da classe `RowNotFoundException` não estão coloridos. Isso ocorre pois, embora essa classe indique que há indícios do IT de persistência, os seus métodos não os apresentam e, portanto, não são coloridos. Os demais métodos não foram alterados, pois nenhum método novo foi identificado.

```
package grenj.persistence;

/**
 * This exception is raised when a row in a table does not exist.
 * @author Vinicius H. S. Durelli<br>
 * date: 30/04/2007 (format = dd/mm/yyyy)
 * @see Error
 * @see Exception
 * @see PersistentObject#load( int, Class )
 */
@SuppressWarnings("serial")
public class RowNotFoundException extends Exception {
```

Figura 5.18: Cabeçalho da classe `RowNotFoundException`.

Finalizado a mineração com o ComSCId, a ferramenta XScan será utilizada com a técnica de mineração por *Concern Peers*. Somente será utilizada a mineração por AspectWiz, pois o autor dessa ferramenta afirma que, por utilizar maior quantidade de técnicas para a criação dos grupos, apresenta resultados com maior probabilidade de implementarem um IT. O resultado obtido com essa configuração pode ser visto na Figura 5.19. Cinco grupos com métodos cujos nomes são interessantes para serem analisados como indícios do IT de persistência são listados, ou seja, os grupos `#0`, `#1`, `#2`, `#3` e `#4`, que agrupam, respectivamente, a implementação dos métodos `insertionValueClause`, `insertionFieldClause`, `updateSetClause`, `deletionClause` e `updateWhereClause`.

Após a análise do código fonte desses métodos, notou-se que eles são os responsáveis por montar o comando SQL a ser utilizado para as atividades de inserção, alteração e exclusão de registros no banco de dados e, portanto, são indícios do IT de persistência. O grupo `#5` não apresenta relação com o IT de persistência e o grupo `#6` apresenta métodos construtores de classes que recebem como parâmetro um objeto `java.sql.ResultSet`, também indícios do IT de persistência. A comparação dos resultados obtidos pela ComSCID com os dos grupos selecionados serve para determinar se todos eles já foram identificados pela ComSCId, ou se foram falsos positivos segundo a técnica de mineração por tipos e textos, com o conjunto de regras atualmente definido.

Se a comparação de resultados não for necessária, sendo que deseja-se exibir todos os

AspectWiz - Peer Groups.txt

```
*** LIST OF PEER GROUPS ***
Number of peer groups: 7

Peer Group #0   Rank: 26.0      Methods: 8              Lowest similarity: 0.5714285714285715
Lgrenj/model/Cash;.insertionValueClause()   Fan-in: 4
Lgrenj/model/CashOnDelivery;.insertionValueClause() Fan-in: 4
Lgrenj/model/EletronicTransfer;.insertionValueClause()  Fan-in: 4
Lgrenj/model/PaymentStrategy;.insertionValueClause()    Fan-in: 4
Lgrenj/model/QualifiableObject;.insertionValueClause()  Fan-in: 3
Lgrenj/model/Resource;.insertionValueClause()   Fan-in: 3
Lgrenj/model/ResourceInstance;.insertionValueClause()   Fan-in: 1
Lgrenj/model/StaticObject;.insertionValueClause()   Fan-in: 3

Peer Group #1   Rank: 26.0      Methods: 8              Lowest similarity: 0.5714285714285715
Lgrenj/model/Cash;.insertionFieldClause()   Fan-in: 4
Lgrenj/model/CashOnDelivery;.insertionFieldClause() Fan-in: 4
Lgrenj/model/EletronicTransfer;.insertionFieldClause()  Fan-in: 4
Lgrenj/model/PaymentStrategy;.insertionFieldClause()    Fan-in: 4
Lgrenj/model/QualifiableObject;.insertionFieldClause()  Fan-in: 3
Lgrenj/model/Resource;.insertionFieldClause()   Fan-in: 3
Lgrenj/model/ResourceInstance;.insertionFieldClause()   Fan-in: 1
Lgrenj/model/StaticObject;.insertionFieldClause()   Fan-in: 3

Peer Group #2   Rank: 14.0      Methods: 8              Lowest similarity: 0.5714285714285715
Lgrenj/model/Cash;.updateSetClause()    Fan-in: 1
Lgrenj/model/CashOnDelivery;.updateSetClause()  Fan-in: 1
Lgrenj/model/EletronicTransfer;.updateSetClause()   Fan-in: 1
Lgrenj/model/PaymentStrategy;.updateSetClause() Fan-in: 1
Lgrenj/model/QualifiableObject;.updateSetClause()   Fan-in: 3
Lgrenj/model/Resource;.updateSetClause()    Fan-in: 3
Lgrenj/model/ResourceInstance;.updateSetClause()    Fan-in: 1
Lgrenj/model/StaticObject;.updateSetClause()    Fan-in: 3

Peer Group #3   Rank: 4.0       Methods: 3              Lowest similarity: 0.5714285714285715
Lgrenj/model/PaymentStrategy;.deletionClause()  Fan-in: 2
Lgrenj/model/ResourceInstance;.deletionClause() Fan-in: 1
Lgrenj/model/StaticObject;.deletionClause() Fan-in: 1

Peer Group #4   Rank: 3.0       Methods: 3              Lowest similarity: 0.5714285714285715
Lgrenj/model/PaymentStrategy;.updateWhereClause()   Fan-in: 1
Lgrenj/model/ResourceInstance;.updateWhereClause()  Fan-in: 1
Lgrenj/model/StaticObject;.updateWhereClause()  Fan-in: 1

Peer Group #5   Rank: 2.0       Methods: 2              Lowest similarity: 1.0
Lgrenj/model/QualifiableObject;.toString()  Fan-in: 1
Lgrenj/model/Resource;.toString()   Fan-in: 1

Peer Group #6   Rank: 0.0       Methods: 4              Lowest similarity: 0.5
Lgrenj/model/Cash;.[constructor](Ljava/sql/ResultSet;Lgrenj/util/Index;)    Fan-in: 0
Lgrenj/model/CashOnDelivery;.[constructor](Ljava/sql/ResultSet;Lgrenj/util/Index;)  Fan-in: 0
Lgrenj/model/EletronicTransfer;.[constructor](Ljava/sql/ResultSet;Lgrenj/util/Index;)   Fan-in: 0
Lgrenj/model/PaymentStrategy;.[constructor](Ljava/sql/ResultSet;Lgrenj/util/Index;) Fan-in: 0
```

Figura 5.19: Resultado obtido pela mineração da XScan.

indícios do IT de persistência sem identificar a ferramenta utilizada, a mesma cor poderia ter sido selecionada para todos os indícios. Desse modo, todos seriam visualizados em conjunto, combinando a precisão e a cobertura das duas técnicas em um único conjunto final de indícios do IT de persistência. Um exemplo desse tipo de resultado é o mostrado na Figura 5.20, que exibe todos os indícios da ferramenta ComSCId e XScan com a coloração verde. Para o exemplo em questão optou-se por colorir, com uma mesma cor, os grupos identificados pela XScan e com cor diferente os indícios da ComSCId, situação apresentada na Figura 5.21.

Outro ponto importante do resultado apresentado pela XScan é que ele não aceita refinamentos. Pode ser interessante saber se esses métodos são invocados nas demais classes do GrenJ, porém, não há como fazer isso quando somente a XScan é usada. Para possibilitar essa visualização a ConcernMapper é usada, sendo que os métodos agrupados pela XScan são utilizados como sementes para que uma análise exploratória seja realizada. A partir dessas se-

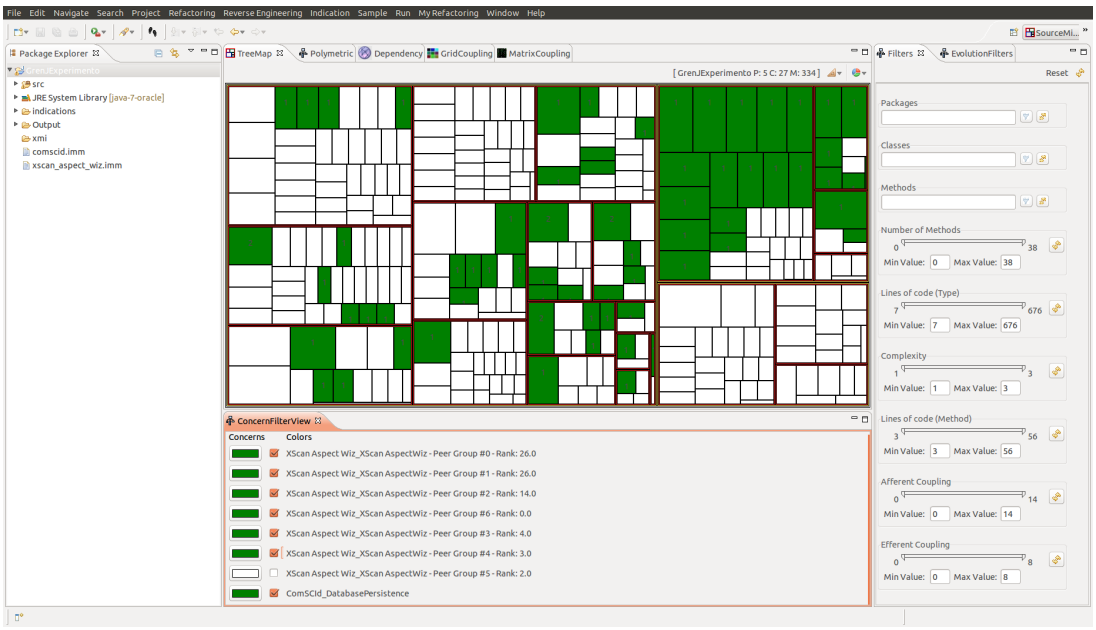


Figura 5.20: Indícios encontrados pela ComSCId e pela XScan combinados pela utilização de uma mesma cor para representá-los.

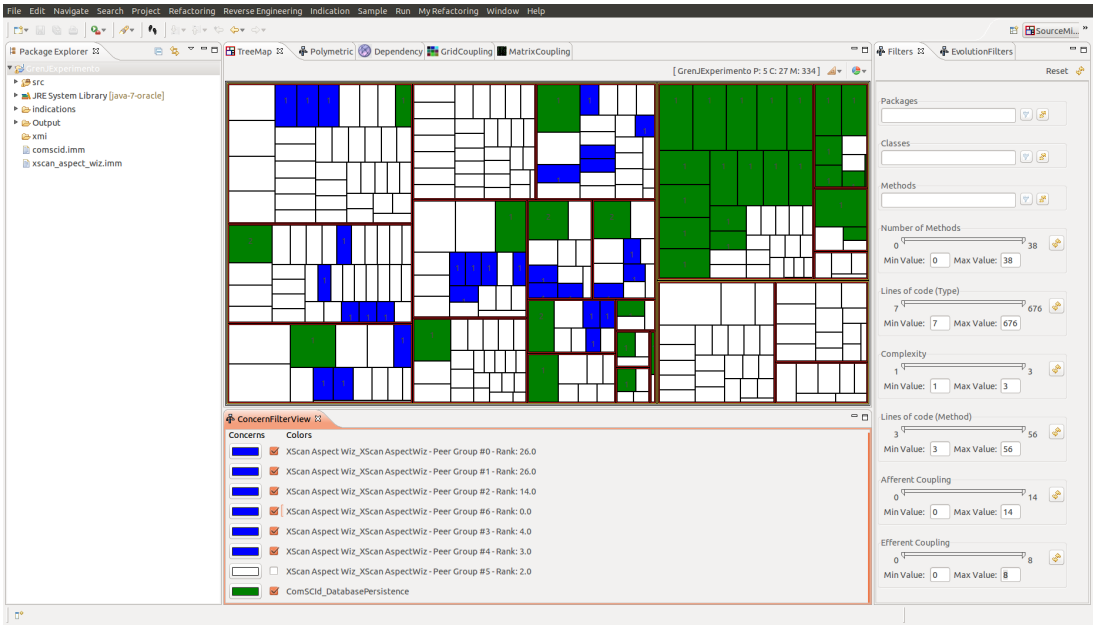


Figura 5.21: Indícios encontrados pela ComSCId e pela XScan com cores diferentes para possibilitar a sua comparação.

mentes, é possível identificar se esses métodos são utilizados em outras estruturas do sistema ou se existem mais métodos como esses que não foram agrupados.

Ao se utilizar a ferramenta ConcernMapper, na área A da Figura 5.22, podem ser vistos alguns dos indícios encontrados pela XScan como sementes de persistência. Na área B pode-se visualizar todas as classes que possuem um método chamado `insertionValueClause()`. Como a ConcernMapper somente realiza buscas de texto, no código fonte, a partir de uma

semente e o nome do método é formado pelo conjunto de caracteres “insertionValueClause”, todos os trechos do código fonte com esse conjunto de caracteres são identificados. Os trechos nos quais ocorre a utilização desse método e também os trechos que declaram esse e outros métodos com esse mesmo nome são identificados como ITs. Ainda nesta área, as classes cujos nomes estão em negrito indicam que o método `insertionValueClause` já foi adicionado como semente na ferramenta ConcernMapper. Com isso pode-se ver que existem outros métodos com o mesmo nome, mas que não foram identificados pela XScan. Isso ocorre também para os outros quatro métodos (`insertionFieldClause`, `updateSetClause`, `deletionClause` e `updateWhereClause`).

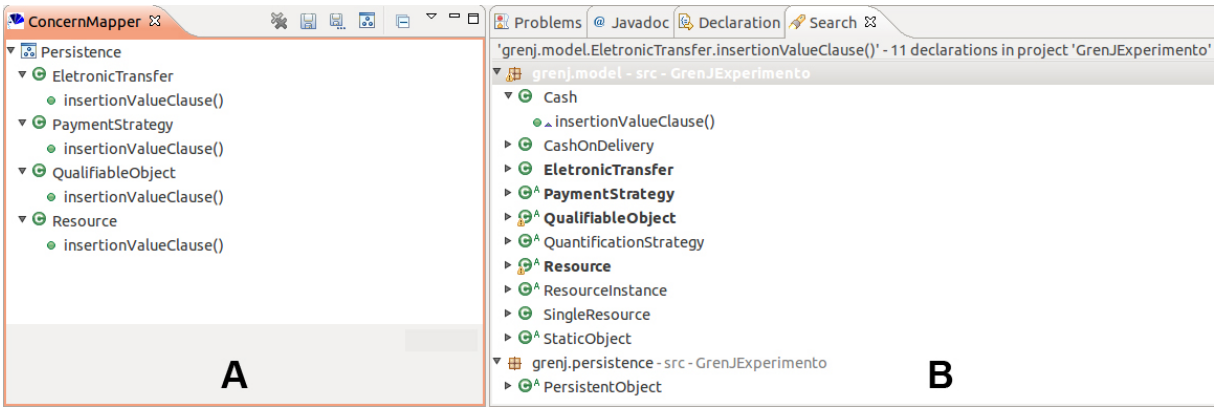


Figura 5.22: Abas da ferramenta ConcernMapper.

Como todos esses métodos têm em comum o conjunto de caracteres “Clause” deve-se criar uma regra para busca por texto, na ComSCId, que procure métodos que contenham esses caracteres em seu nome, assim facilitando a identificação de todos esses métodos. Esse novo resultado será detalhado mais a adiante, quando forem comparados os resultados obtidos pela XScan e pela ComSCId, para verificar se essa nova regra realmente identificou corretamente todos esses indícios.

Dando continuidade ao processo de mineração com a ferramenta XScan, é necessário realizar a análise do grupo `#6`, composto por construtores de classes que recebem como parâmetro um objeto do tipo `java.sql.ResultSet`. Para facilitar a visualização, na Figura 5.23 é mostrado um desses métodos. Nesse caso, pode-se visualizar um número 2 dentro do retângulo que o representa e a sua descrição, indicando que esse método foi identificado tanto pela XScan quanto pela ComSCId. Esse tipo de identificação ocorre para todos os métodos desse grupo e, portanto, nenhuma análise futura é requerida, visto que todos foram identificados pela ComSCId. A partir desse momento, esse grupo não será mais colorido, pois não há a necessidade dessa duplicação de informação.

Para finalizar a análise utilizando a ferramenta XScan é necessário verificar se todos os

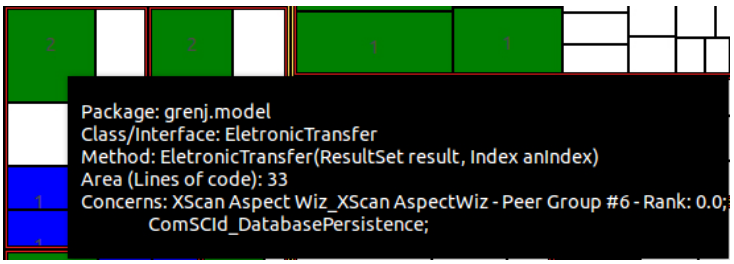


Figura 5.23: Descrição de um dos métodos do grupo `#6` identificados pela XScan.

indícios dos grupos `#0`, `#1`, `#2`, `#3` e `#4` foram identificados pela ComSCId após a criação da regra "Clause", comentada anteriormente. Esse resultado pode ser visto na Figura 5.24, na qual somente a cor verde aparece, indicando que os indícios identificados pela ComSCId sobrepuseram todos os métodos identificados pela XScan (coloridos com a cor azul). Sendo assim, os indícios da XScan não são mais utilizados, evitando uma sobrecarga de informações na listagem de ITs a serem coloridos

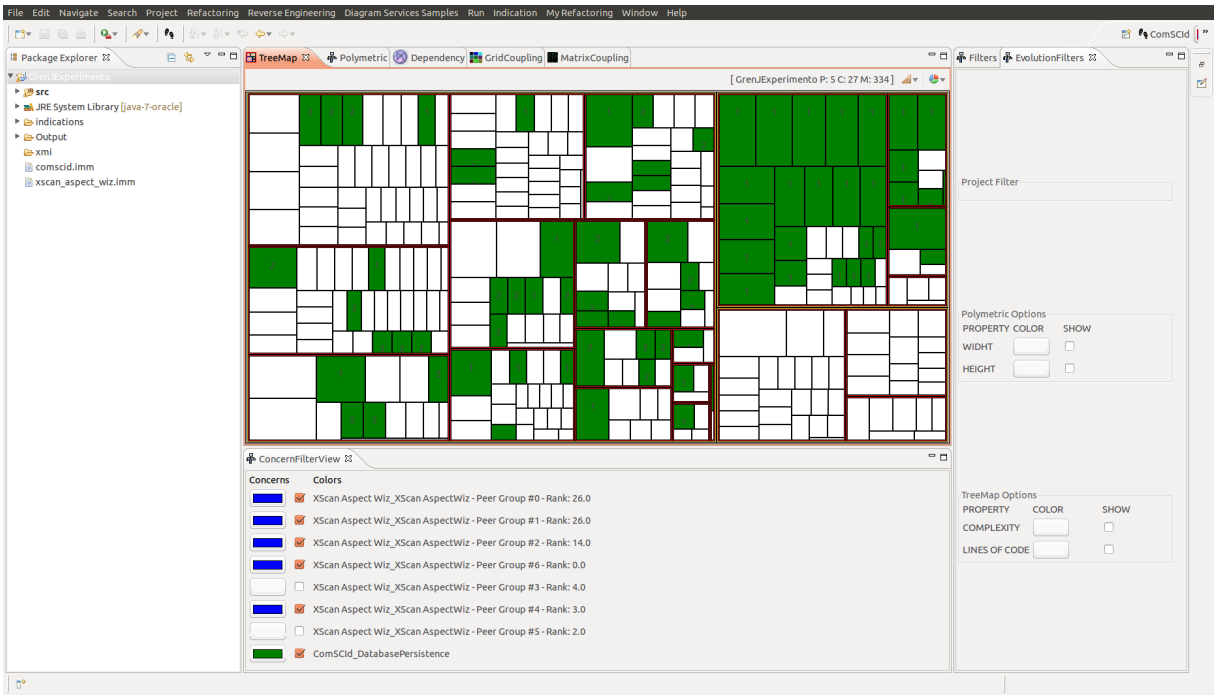


Figura 5.24: Resultado obtido após a criação de regra para identificar todos os métodos cujos nomes contém o conjunto de caracteres "Clause".

Como atividade final da prova de conceito, deve-se utilizar a análise por *Fan-in* disponibilizada pela ferramenta FINT. Para isso utilizou-se um *Threshold* igual a 5 e os métodos do sistema não foram filtrados. Os resultados obtidos podem ser visualizados na Figura 5.25. Após uma análise detalhada desses métodos pôde-se identificar quais representam indícios do IT de persistência e, portanto, devem ser coloridos de forma a complementar a análise realizada até o momento. Para facilitar a visualização, esses métodos foram destacados por quadrados ver-

melhos na própria Figura 5.25. Dos métodos identificados pela FINT, vale destacar os métodos `append`, `toString()` e o construtor da classe `StringBuilder` que representam indícios do IT de persistência por serem somente utilizados, nesse sistema, para a criação dos comandos SQL.

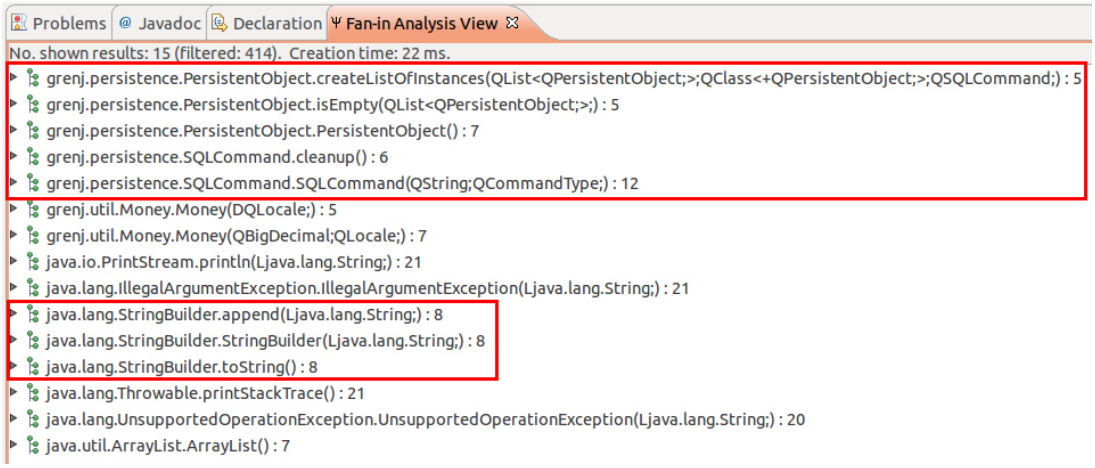


Figura 5.25: Aba da ferramenta FINT com os métodos identificados.

Na Figura 5.26 é mostrado o resultado final da mineração do IT de persistência no sistema GrenJ. Neste momento não há a necessidade de comparação entre os indícios encontrados pelas ferramentas ComSCId e FINT, visto que o objetivo é somente visualizar todos os indícios encontrados. Por esse motivo, tanto os indícios da ComSCId, quanto da FINT foram coloridos com a mesma cor, verde.

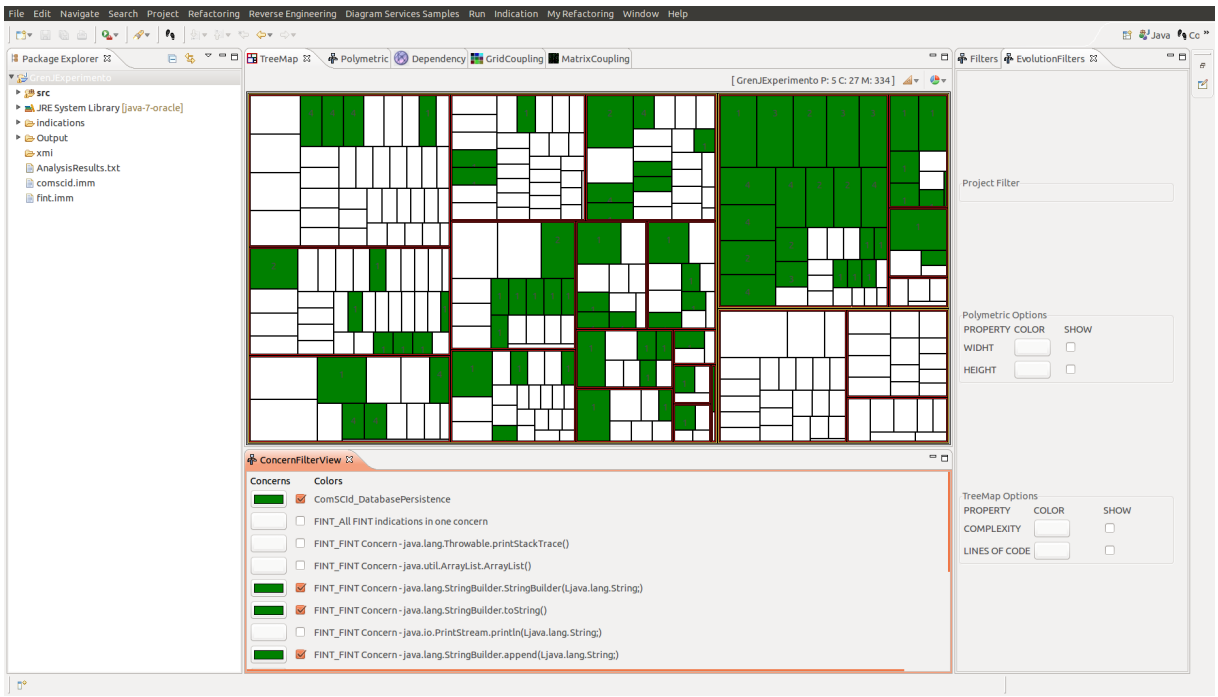


Figura 5.26: Resultado final da mineração do IT de persistência.

Vale destacar que a técnica de mineração utilizada pela FINT, por utilizar a métrica *fan-in*

para identificar os indícios de ITs, pode apresentar em seu resultado os indícios de diferentes ITs. Isso demanda um maior esforço, por parte do engenheiro de software, para analisar esse resultado, porém torna essa ferramenta em uma boa candidata a ferramenta inicial do processo de mineração, assim como a ComSCId. Isso ocorre pois, o seu resultado apresenta os diversos possíveis ITs que afetam o sistema e pode ser utilizado como uma base inicial de sementes na ferramenta ConcernMapper.

Ao analisar novamente os resultados apresentados na Figura 5.25 pode-se perceber alguns métodos do pacote `java.lang.Throwable`, que representam indícios do IT tratamento de exceções. Para facilitar a visualização, esses métodos encontram-se destacados na Figura 5.27. A partir desses métodos identificados pode-se iniciar uma nova análise do sistema, utilizando novamente todas as ferramentas de mineração. A Figura 5.28 apresenta todos os indícios do IT de tratamento de exceção coloridos com a cor amarela.

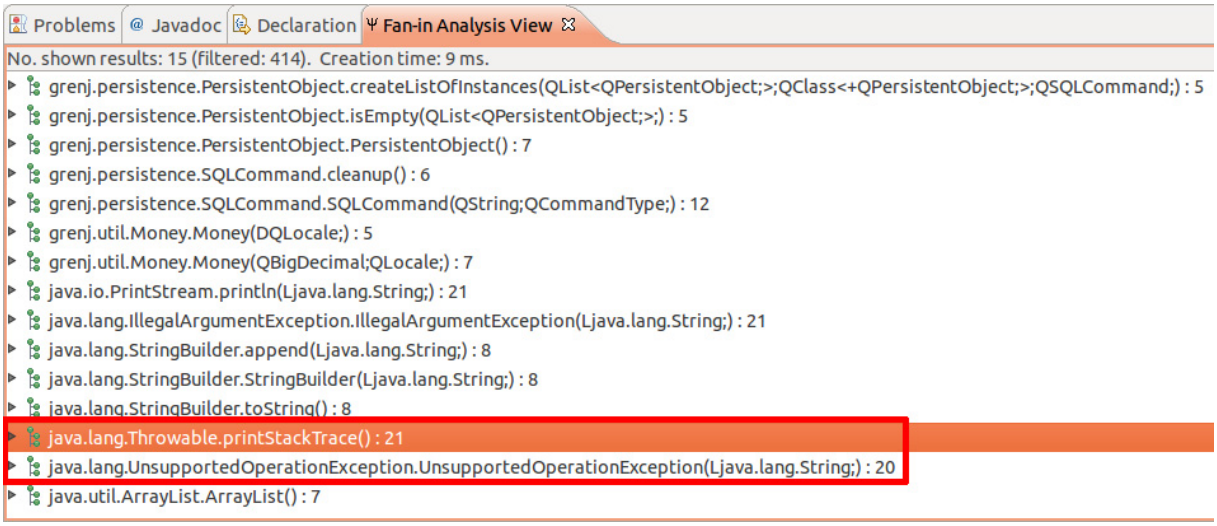


Figura 5.27: Aba da ferramenta FINT com indícios do IT de tratamento de exceção destacados.

5.4 Considerações Finais

As ferramentas de MIT, por terem enfoque na descoberta dos indícios de ITs, em alguns casos não apresentam uma forma satisfatória de visualização de seus resultados. Um exemplo disso é a XScan (NGUYEN *et al.*, 2011) que utiliza de um arquivo texto, o que dificulta a análise geral dos indícios encontrados e como eles afetam a estrutura do software. Uma forma de visualizar melhor esses resultados é por meio das visões das ferramentas de visualização de software. Para isso é necessário integrar essas ferramentas, o que não é trivial, porém possibilita a combinação de seus pontos positivos.

Cada ferramenta de MIT possui um arquivo de saída, contendo os indícios encontrados

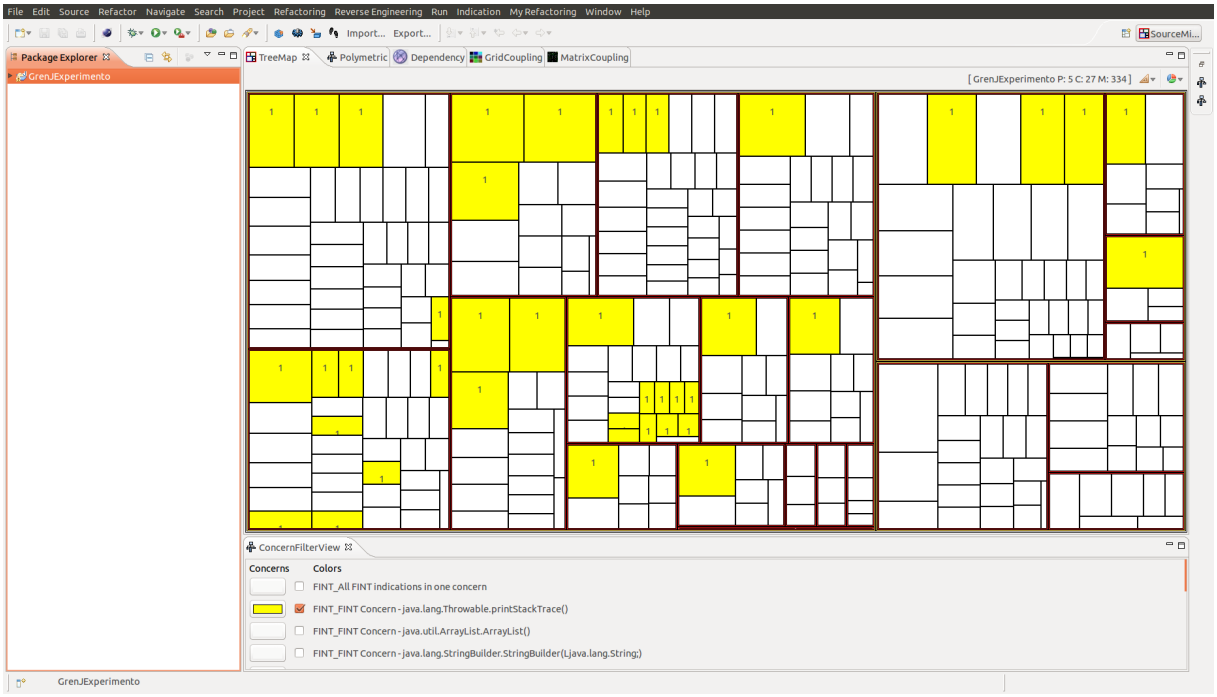


Figura 5.28: Indícios do IT de tratamento de exceção identificados pela ferramenta FINT

durante a análise realizada, que segue padrões próprios, com particularidades e pouca documentação. O metamodelo proposto tem como objetivo minimizar esse esforço de utilização de ferramentas de MIT e visualização de software combinadas, criando uma forma comum de identificação dos indícios encontrados pelas ferramentas de MIT e padronizando a forma de comunicação entre essas ferramentas.

Comparar os resultados obtidos pelas ferramentas de MIT sem o auxílio de uma ferramenta não é simples, mas apresenta a vantagem de evidenciar fortemente os falsos negativos (CECCATO *et al.*, 2006). As ferramentas de visualização, ao utilizarem o metamodelo proposto, podem exibir esses resultados por meio de suas visões, facilitando sua comparação e possibilitando a identificação de novos padrões visuais que evidenciam estruturas com maior probabilidade de representarem falsos positivos. Dessa forma, a precisão de diversas técnicas de MIT é combinada, de modo a diminuir a ocorrência de falsos negativos e positivos, melhorando o resultado final apresentado. Apesar de o IMM facilitar a utilização em conjunto dessas ferramentas, não é facilitada a utilização individual de cada uma delas, sendo necessário dominar a sua utilização individual, para beneficiar-se da integração.

De forma a avaliar os benefícios proporcionados pelo IMM foram propostos dois experimentos: i) avaliar a utilização do metamodelo na integração de ferramentas de MIT e de visualização de software e; ii) avaliar se a utilização em conjunto dessas ferramentas apresenta um melhor resultado, quando comparado com sua utilização individual. Esses experimentos

foram planejados de acordo com passos definidos por Wohlin *et al.* (2000) e serão apresentados no próximo capítulo.

Capítulo 6

EXPERIMENTOS SOBRE A INTEGRAÇÃO E A UTILIZAÇÃO DAS FERRAMENTAS MIT E DE VISUALIZAÇÃO DE SOFTWARE

6.1 Considerações Iniciais

Uma das formas de verificar os resultados obtidos em uma pesquisa é por meio da realização de experimentos, que proporciona a utilização de testes estatísticos para a inferência de hipóteses (WOHLIN *et al.*, 2000). Desse modo, os experimentos apresentados neste capítulo seguem os passos propostos por Wohlin *et al.* (2000), que podem ser divididos em:

- **Definição** dos objetivos do experimento;
- **Planejamento** de como o experimento deve ser conduzido e;
- **Operação**, que apresenta a realização do experimento e a análise dos dados coletados.

O primeiro experimento realizado avaliou o apoio oferecido pelo metamodelo *Indications Metamodel* (IMM) na integração de ferramentas de mineração de interesses transversais (MIT) com as de visualização de software. Neste experimento alunos de pós-graduação em Ciência da Computação da Universidade Federal de São Carlos realizaram a integração da ferramenta ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a) com a ferramenta SourceMiner (CARNEIRO *et al.*, 2010), utilizando arquivos de integração, que seguem dois formatos: o IMM e o CM. O formato IMM é uma das contribuições deste trabalho e o CM é o formato apresentado pela ferramenta ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005), já utilizado para integrá-la à SourceMiner.

O segundo experimento realizado avaliou o apoio oferecido pelo ambiente integrado de ferramentas MIT e visualização de software na identificação dos indícios de ITs existentes em um software. Esse experimento foi realizado com alunos de graduação em Ciência da Computação da Universidade Federal de São Carlos, que analisaram dois sistemas diferentes, a fim de identificar os indícios de ITs com e sem o auxílio do ambiente integrado.

O experimento de utilização do IMM é descrito na Seção 6.2 e o de avaliação do ambiente integrado é descrito na Seção 6.3. Na Seção 6.4 são apresentadas as considerações finais deste capítulo.

6.2 Experimento de Utilização do IMM

Nesta seção é apresentado um experimento para observar a utilização do IMM e quanto com o seu uso fica facilitada a integração das ferramentas de mineração de interesses transversais e de visualização de software. A ferramenta de MIT utilizada foi a ComSCId e a de visualização foi a SourceMiner, com o formato de integração IMM e CM (utilizado pela ConcernMapper).

O experimento foi planejado de acordo com os passos propostos por Wohlin *et al.* (2000), que são descritos a seguir.

6.2.1 Definição

Analisar: o apoio fornecido pelo formato de integração IMM
Com o propósito de: comparar com o apoio fornecido pelo formato CM
Com respeito ao: tempo total gasto e a quantidade de erros cometidos durante a realização da integração
Do ponto de vista: do desenvolvedor
No contexto de: alunos de pós-graduação em Ciência da Computação da Universidade Federal de São Carlos.

6.2.1.1 Objetos de Estudo

Os objetos de estudo são os formatos de integração IMM e CM.

6.2.1.2 Objetivo

O objetivo é comparar o apoio fornecido pelo o formato de integração IMM com o fornecido pelo formato CM. O formato IMM proposto neste trabalho, tem como enfoque representar os indícios de ITs presentes no código fonte de um software.

O CM é o formato do arquivo gerado pela ferramenta ConcernMapper, o qual já estava integrada a SourceMiner. Neste experimento optou-se por utilizar uma versão adaptada desse formato, sendo que algumas regras de conversão dos formatos Java para o formato CM foram omitidas dos participantes. Essa ação foi tomada para não sobrecarregar os participantes com as particularidades desse formato.

6.2.1.3 Enfoque Quantitativo

O enfoque quantitativo analisa o tempo total gasto e os erros cometidos por um participante para realizar a integração entre as ferramentas ComSCId e SourceMiner utilizando o formato IMM e o CM.

6.2.1.4 Perspectiva

O experimento foi realizado sob a perspectiva do desenvolvedor que deseja integrar sua ferramenta de MIT com uma ferramenta de visualização de software.

6.2.1.5 Contexto

O experimento foi realizado por alunos de pós-graduação em Ciência da Computação da Universidade Federal de São Carlos, que concordaram em participar do experimento ao assinarem um termo de consentimento análogo ao do Apêndice C.

6.2.2 Planejamento

Esse experimento foi planejado de forma a responder a duas questões de pesquisa: Q1: “A utilização do IMM diminui a quantidade de tempo necessária para realizar a integração de uma ferramenta de MIT com uma de visualização de software?” e; Q2: “A utilização do IMM diminui a quantidade de erros cometidos na geração do arquivo de integração?”.

Todos os participantes realizaram a integração da ferramenta ComSCId com a SourceMiner por meio do formato IMM e CM, realizando somente as atividades de conversão dos dados,

com a disponibilização prévia das funções para escrita dos arquivos. Para responder a primeira questão foi medida a quantidade de tempo total utilizada pelo participante para finalizar a conversão dos dados. Para responder a segunda questão, os arquivos de integração gerados foram analisados de forma a quantificar o número de erros existente em sua estrutura e informações. Erros de estrutura são referentes à forma como os dados devem ser apresentados, como por exemplo, em um arquivo XML deve ter seu cabeçalho e a abertura e fechamento de suas *tags*. Já erros de informação são aqueles cujos dados do arquivo não foram corretamente convertidos.

6.2.2.1 Seleção do contexto

O experimento foi realizado com dez alunos de pós-graduação em Ciência da Computação, em Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos (UFSCar), no segundo semestre de 2013. Todos os participantes possuíam conhecimento prévio de Orientação a Objetos e da linguagem de programação Java.

6.2.2.2 Formulação das Hipóteses

As questões de pesquisa foram formuladas da seguinte forma:

- Q1, Hipótese Nula (H_0): considerando os formatos IMM e CM, não há diferença significativa entre o tempo total necessário para integrar as ferramentas de MIT e de visualização de software;

$$H_0 : \mu_{IMM} = \mu_{CM}$$

- Q1, Hipótese Alternativa (H_1): o tempo necessário para realizar a integração utilizando o formato IMM é significativamente menor que o tempo necessário para realizar a integração utilizando o formato CM.

$$H_1 : \mu_{IMM} < \mu_{CM}$$

- Q2, Hipótese Nula (H_0): não há diferença significativa entre a quantidade de erros presentes no conteúdo dos arquivos IMM e CM gerados durante a integração;

$$H_0 : \mu_{IMM} = \mu_{CM}$$

- Q2, Hipótese Alternativa (H_1): a quantidade de erros no conteúdo do arquivo IMM é significantemente menor que a quantidade de erros no conteúdo do arquivo CM.

$$H_1 : \mu_{IMM} < \mu_{CM}$$

6.2.2.3 Seleção das Variáveis

As variáveis dependentes consideradas neste experimento foram: "tempo necessário para realizar a análise" e "número de erros existente no conteúdo dos arquivos IMM e CM gerados". As variáveis independentes consideradas foram:

- **Aplicações:** todo participante teve que integrar a ferramenta ComSCId com a SourceMiner. A ComSCId é uma ferramenta de MIT, que utiliza a técnica de mineração de análise por tipos e texto. A SourceMiner é uma ferramenta de visualização permite sua integração tanto por IMM quanto por CM;
- **Ambiente de Desenvolvimento:** Eclipse 3.6.2 (Eclipse Foundation, 2013);
- **Tecnologias:** Java version 7.

6.2.2.4 Seleção dos Participantes

Os participantes foram selecionados por meio de uma amostragem não probabilística e por conveniência, ou seja, não é conhecida a probabilidade correspondente dos elementos de toda a população em relação a essa amostra.

6.2.2.5 Projeto do Experimento

Os participantes não foram divididos em grupos, sendo que todos os participantes realizaram a integração por meio do IMM e depois por meio do CM. Optou-se por não dividir os participantes em grupos, pois havia somente duas atividade a realizar, sendo integrar a ComSCId e a SourceMiner utilizando o formato IMM e o CM.

6.2.2.6 Projeto dos Tipos

O experimento foi estruturado de acordo com o projeto "um fator com dois tratamentos pareados" (WOHLIN *et al.*, 2000). O fator desse experimento refere-se à maneira como o IMM

foi utilizado para integrar a ComSCId e a SourceMiner. Os tratamentos referem-se ao apoio oferecido pelo IMM comparado ao oferecido pelo CM.

6.2.2.7 Instrumentação

Todos os materiais necessários para a realização do experimento foram previamente entregues aos participantes. Esse material era composto por: i) formulário para coletar dados experimentais (Apêndice A), neste caso, tempo necessário para realizar a integração; ii) diagrama de classes do IMM e; iii) descritivo das regras de criação de um arquivo CM (Apêndice B).

6.2.3 Operação

A fase de operação foi dividida em duas partes, como apresentado nas próximas subseções:

6.2.3.1 Preparação

Os participantes receberam treinamento sobre o código fonte da ferramenta ComSCId, sendo explicada a sua estrutura de classes, de modo que soubessem recuperar as informações a serem convertidas. Depois foi apresentada a ferramenta SourceMiner, mostrando suas formas de visualização de um software e os passos necessários para testar a integração das ferramentas.

6.2.3.2 Execução

Primeiramente, os participantes assinaram o formulário de consentimento. Em seguida receberam um treinamento sobre a ComSCId, a SourceMiner e como realizar a integração de ferramentas por meio de: i) serialização de classes em arquivo XML e; ii) criação de arquivos de texto com um formato específico.

No treinamento sobre a ComSCId foi apresentada a técnica de MIT por tipos e texto, explicando para os participantes como realizar uma análise a fim de recuperar os indícios de ITs a serem convertidos. Também foi apresentada a organização de seu código fonte, explicando sobre as estruturas da ComSCId a serem convertidas e como recuperar essas informações. Quanto ao treinamento da SourceMiner, foi apresentado o seu funcionamento, sendo definido como deveria ser testada a integração. Por fim foi apresentado como funciona um processo de conversão dos dados da ComSCId, por meio da serialização de um conjunto de classes em um arquivo

XML e por meio da criação de um arquivo texto. Os formatos utilizados no treinamento eram diferentes do IMM e do CM, porém eram semelhantes em nível de complexidade.

O piloto do experimento foi executado logo após a finalização do treinamento. O piloto tinha como objetivo representar um experimento real, para que dúvidas quanto aos passos a serem realizados e o preenchimento dos formulários não influenciassem na tomada de tempo durante o experimento. Antes da execução do piloto, somente foi explicado como preencher o formulário de coleta de dados experimentais, não sendo dada nenhuma informação sobre os formatos de integração.

Tendo os participantes entendido o formulário de coleta de dados e preparado seus ambientes, foi iniciada a tomada de tempo para a integração. Todos os participantes tiveram que integrar a ComSCid com a SourceMiner, utilizando os formatos criados para o piloto, diferentes do IMM e do CM, porém com complexidade semelhante.

6.2.4 Análise dos Dados

Esta seção apresenta os resultados estatísticos dos dados coletados do experimento. A análise está dividida em duas subseções: Estatística Descritiva (WOHLIN *et al.*, 2000) e Teste de Hipótese (WOHLIN *et al.*, 2000). A primeira visa descrever o conjunto de dados coletados da amostra por meio de técnicas estatísticas, visando identificar dados que possam interferir na obtenção dos resultados e informações necessárias para decidir quais técnicas de validação das hipóteses devem ser utilizadas. A Seção 6.2.4.2 apresenta a inferência das hipóteses formuladas a partir das questões de pesquisa.

6.2.4.1 Estatísticas Descritivas

Os dados coletados podem ser vistos na Tabela 6.1. A primeira coluna contém os participantes do experimento listados um em cada linha. A segunda coluna apresenta o tempo gasto em minutos pelos participantes para gerar o arquivo de integração, sendo dividida entre o formato IMM e o formato CM. A terceira coluna apresenta a quantidade de erros presentes no conteúdo dos arquivos gerados. A penúltima linha dessa matriz expressa a média dos valores para cada uma das colunas e a última linha apresenta a porcentagem que essa média representa em relação ao tempo total médio, ou a quantidade de erros total média. Ao analisar o valor médio e a porcentagem do tempo utilizado pelos participantes, ilustrados na penúltima e ultima linhas, respectivamente, pode-se notar diferença significativa entre o tempo gasto na utilização do IMM (22%), em comparação com o tempo gasto na utilização do CM (78%). Também

Tabela 6.1: Dados coletados no experimento de integração usando os formatos IMM e CM.

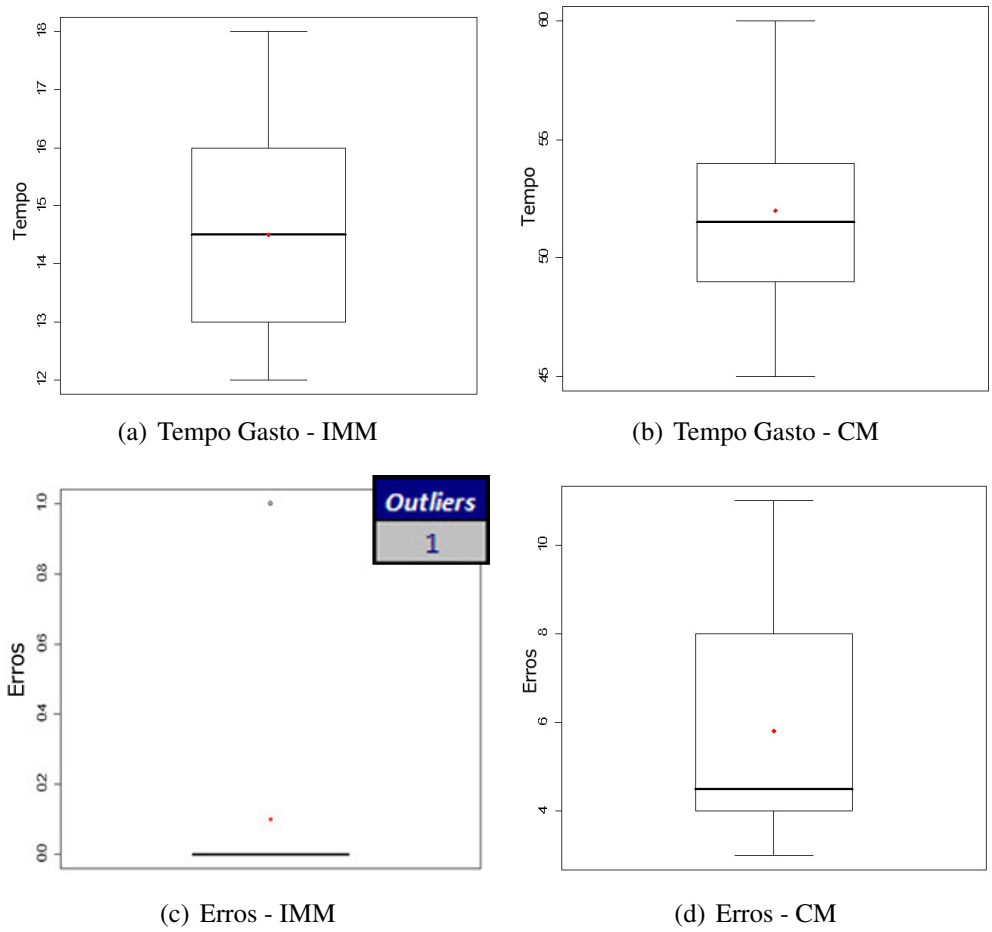
	Tempo Gasto		Erros	
Participante	IMM	CM	IMM	CM
P1	13	60	1	11
P2	12	54	0	5
P3	18	54	0	3
P4	14	48	0	4
P5	15	45	0	7
P6	17	57	0	3
P7	13	50	0	9
P8	12	53	0	4
P9	16	49	0	8
P10	15	50	0	4
AVG	14.6667	52	0.1	5.8
%	22	78	1.69492	98.30508

pode-se notar diferença expressiva entre os erros apresentados durante a utilização do IMM (1,69%) e do CM (98,31%). Desses resultados vale destacar o participante P1, que foi o único a apresentar um erro no conteúdo do arquivo IMM, isso ocorreu em razão de um engano na lógica implementada por ele, que ao invés de converter o nome de uma estrutura do ComSCId converteu seu `ID`.

Primeiramente deve-se verificar a existência de *outliers*, que apresentam um comportamento extremo, ou seja, apresentam comportamento diferente dos demais (ESTATCAMP, 2013d). Para isso foi utilizado o gráfico de *boxplot*, que exibe a distribuição empírica dos dados (ESTATCAMP, 2013a). Esses grafos foram obtidos com o auxílio de um software de estatística chamado Action (ESTATCAMP, 2013e), que é integrado ao Excel (MICROSOFT, 2013), e podem ser visualizados na Figura 6.1. No gráfico apresentado na Figura 6.1(c) está representada a distribuição dos valores coletados quanto aos erros existentes no arquivo de integração gerado pelo formato IMM. Como comentado anteriormente, observa-se que o participante P1 é um *outlier* e, portanto, deve ser removido da análise de inferência das hipóteses.

O próximo passo é decidir se o conjunto de dados provêm ou não de uma distribuição normal. Essa informação é necessária para decidir o teste de inferência a ser aplicado para avaliar as hipóteses. Para essa análise decidiu-se utilizar o teste de Shapiro-Wilk (ESTATCAMP, 2013b), aplicado com o auxílio do software Action. Todas as colunas da Tabela 6.1 foram analisadas, uma a uma, e os resultados podem ser vistos na Figura 6.2.

O conceito de P-valor indica a probabilidade que o valor estatístico de um teste tenha valor extremo, quando a hipótese nula for verdadeira (ESTATCAMP, 2013c). Em outras palavras, o P-valor indica a probabilidade de a hipótese nula ocorrer. Ao se comparar o P-valor com o nível de significância (α) pode-se ou não rejeitar a hipótese nula. Quando P-valor é menor que α, rejeita-se a hipótese nula, pois a probabilidade dela ocorrer é muito baixa. Caso contrário ela é aceita, pois a probabilidade dela ocorrer é maior do que α. O nível de significância utilizado



(a) Tempo Gasto - IMM (b) Tempo Gasto - CM

(c) Erros - IMM (d) Erros - CM

Figura 6.1: Boxplots dos dados da Tabela 6.1.

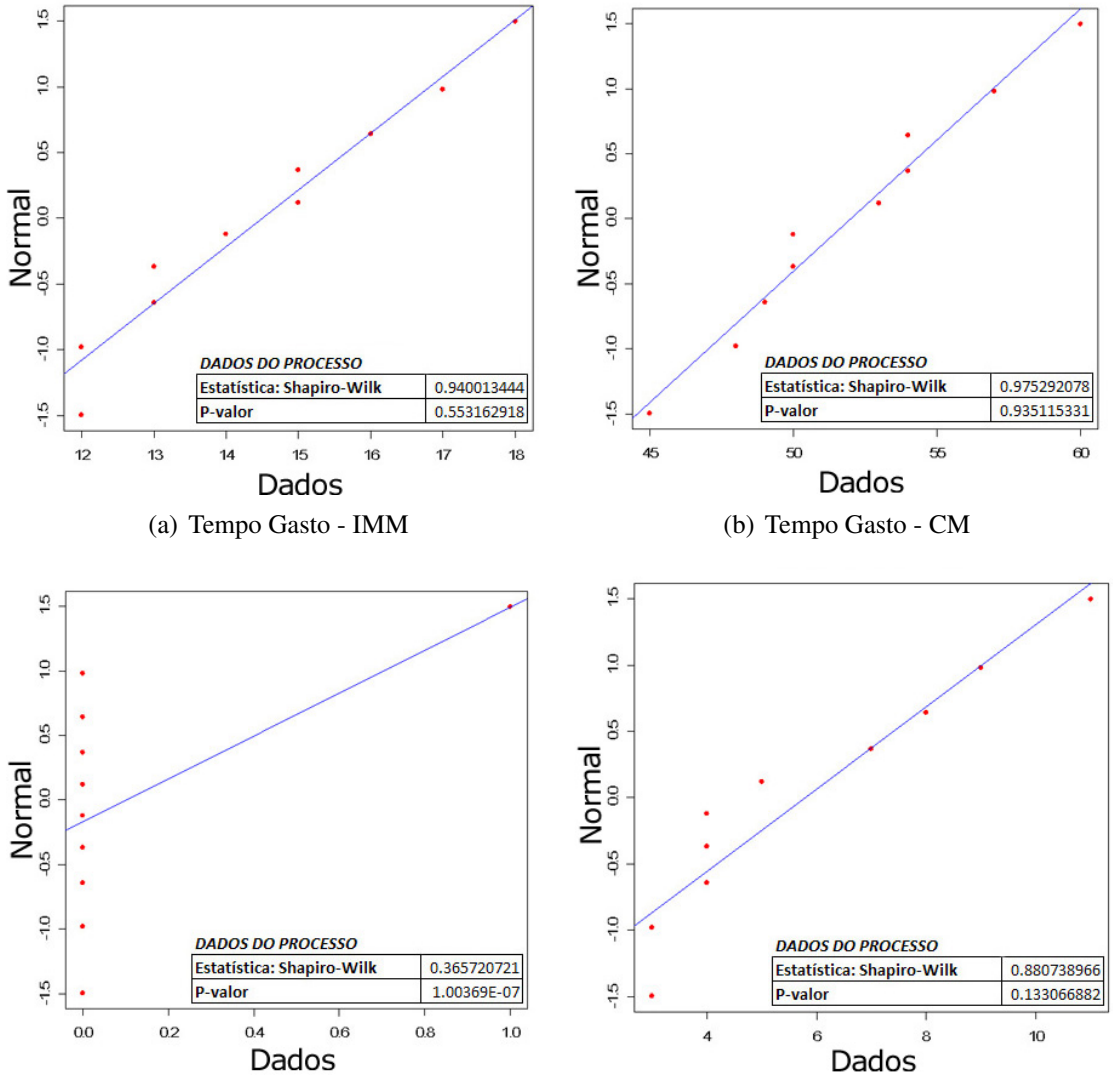
para análise dos dados deste experimento é 95%, ou seja, $\alpha = 0,05$.

O teste de Shapiro-Wilk propõe que a hipótese nula é: "A amostra provém de uma população Normal" (ESTATCAMP, 2013b). Ao analisarmos os P-valores apresentados na Figura 6.2, nota-se que somente os valores dos erros apresentados na Figura 6.2(c) provêm de uma distribuição não normal. Com isso, para a inferência das hipóteses, foi utilizado o Teste T pareado[1] na análise dos dados de tempo e o teste de Wilcoxon[2] na análise dos dados dos erros.

6.2.4.2 Testando as Hipóteses

Como comentado no final a seção anterior serão usados o Teste T Pareado e o teste de Wilcoxon Pareado. Esses testes foram aplicados com o auxílio do software Action, a fim de avaliar as hipóteses das duas questões de pesquisa. Os resultados são apresentados na Figura 6.3.

[1] http://www.portalaction.com.br/en/615-t-test-paired
[2] http://www.portalaction.com.br/974-3-teste-de-wilcoxon-pareado



(a) Tempo Gasto - IMM

(b) Tempo Gasto - CM

(c) Erros - IMM

(d) Erros - CM

Figura 6.2: Testes de normalidade dos dados da Tabela 6.1.

Para o tempo, apresentado na Figura 6.3(a), P-valor é aproximadamente 10^{-9}, que é menor do que $\alpha = 0,05$. Portanto, para um nível de significância de 95% pode-se rejeitar a hipótese nula, aceitando-se a hipótese alternativa que é: "o tempo necessário para realizar a integração utilizando o formato IMM é significativamente menor que o tempo necessário para realizar a integração utilizando o formato CM".

Para a análise dos erros cometidos, apresentada na Figura 6.3(b), o P-valor é aproximadamente $0,005$, que é menor que $\alpha = 0,05$. Portanto, para um nível de significância de 95%, pode-se rejeitar a hipótese nula, aceitando-se a hipótese alternativa que é: "a quantidade de erros no conteúdo do arquivo IMM é significantemente menor que a quantidade de erros no conteúdo do arquivo CM".

Informação	Valor
T	-27.52514855
Graus de Liberdade	8
P-valor	1.63654E-09
Média das Diferenças	-36.44444444
Desvio Padrão das diferenças	3.972125096
Hipótese Alternativa: Menor que	0
Intervalo de Confiança	95%
Limite Superior	-33.9823253

(a) Teste T Pareado - Tempo

Informação	Valor
V	0
P-valor	0.004424387
Hipótese Nula	0
Método	Wilcoxon signed rank test with continuity correction
(Pseudo) Mediana	-5.000026585
Intervalo de Confiança - Limite Superior	-3.500049647

(b) Wilcoxon Pareado - Erros

Figura 6.3: Testes de Inferência

6.2.5 Ameaças à Validade

É importante considerar o quão válidos os resultados são, sendo assim, esta seção apresenta algumas das ameaças que podem afetar o valores coletados e a conclusão obtida.

1. Validade interna

   - O nível de conhecimento dos participantes era bastante variado. Visando amenizar essa ameaça os participantes receberam treinamento prévio sobre as ferramentas utilizadas e como realizar a integração de ferramentas utilizando as técnicas de serialização de classes em arquivo XML e criação de arquivos texto com formato específico;
   - Computadores e instalações diferentes podem afetar o resultado. Visando amenizar essa ameaça, o experimento foi realizado em Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos (UFSCar);
   - O receio dos participantes de que seriam avaliados de acordo com os resultados do experimento. Visando amenizar essa ameaça, foi claramente explicado que o experimento não seria, de forma alguma, utilizado para avaliá-los.

2. Validação pela Construção

   - Os participantes conheciam os pesquisadores e sabiam que a utilização do IMM como forma de integração poderia ser mais rápida, o que poderia influenciar os resultados, tornando o experimento menos imparcial. Para amenizar essa ameaça, conduziu-se o experimento de forma que os participantes mantivessem um ritmo constante de desenvolvimento.

3. Validade Externa

- É possível que as atividades propostas durante o experimento não representem situações do mundo real. Para amenizar essa ameaça, foram selecionadas uma ferramenta de MIT e uma de visualização de software, dentre as ferramentas integradas durante a execução deste trabalho.

4. Validade da Conclusão

- Os resultados obtidos são influenciados diretamente pela precisão das métricas utilizadas durante a execução do experimento. Visando amenizar essa ameaça, utilizou-se somente o tempo total gasto pelos participantes durante o desenvolvimento da integração e a quantidade de erros apresentados no arquivo final de integração gerado;
- A quantidade de participantes foi pequena, o que apresenta um baixo poder estatístico. Visando suavizar essa ameaça, aplicou-se o Teste T Pareado e o Teste de Wilcoxon Pareado.

6.3 Experimento de Utilização do Ambiente Integrado

Nesta seção é apresentado um experimento para observar o apoio oferecido pelo ambiente integrado de ferramentas MIT e visualização de software, descrito no Capítulo 5, na identificação dos indícios de ITs existentes em um software. Esse ambiente integrado possibilita que as ferramentas de MIT ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005), ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a), FINT (MARIN *et al.*, 2007), XScan (NGUYEN *et al.*, 2011) sejam utilizadas em conjunto com a ferramenta de visualização de software chamada SourceMiner (CARNEIRO *et al.*, 2010).

O experimento foi planejado de acordo com os passos propostos por Wohlin *et al.* (2000), que são descritos a seguir.

6.3.1 Definição

Analisar: os ambientes integrados e não integrados de ferramentas de MIT e visualização de software
Com o propósito de: comparar a utilização desses ambientes
Com respeito ao: tempo total gasto para a análise e a assertividade do resultado final obtido
Do ponto de vista: engenheiro de software
No contexto de: alunos de graduação em Ciência da Computação da Universidade Federal de

São Carlos.

6.3.1.1 Objetos de Estudo

Os objetos de estudo são os ambientes integrado e não integrado de ferramentas de MIT e de visualização de software.

O ambiente integrado foi descrito na prova de conceito apresentada no Capítulo 5, no qual as ferramentas de MIT ConcernMapper, ComSCId, FINT e XScan foram utilizadas em conjunto com a ferramenta de visualização SourceMiner. O ambiente não integrado, refere-se a utilização dessas mesmas ferramentas, porém os resultados obtidos pelas ferramentas de MIT não podem ser analisados por meio das visões da SourceMiner.

6.3.1.2 Objetivo

O objetivo pretendido foi analisar se a utilização de ferramentas de MIT integradas a uma ferramenta de visualização de software diminuiu o tempo necessário para realizar a análise. Também é analisado se o resultado final apresenta uma maior quantidade de indícios de ITs identificados e uma menor quantidade de falsos negativos e falsos positivos.

6.3.1.3 Enfoque Quantitativo

O enfoque quantitativo analisa o tempo total gasto por um participante, para analisar um software à procura dos indícios do IT de persistência, e a quantidade total de indícios, de falsos negativos e de falsos positivos presentes no resultado final dessa análise.

O IT de persistência foi escolhido por ser um IT bastante comum, presente em grande parte dos sistemas computacionais. Com isso esperava-se que os participantes tivessem um maior conhecimento prévio desse IT, facilitando a execução do experimento.

6.3.1.4 Enfoque Qualitativo

O enfoque qualitativo analisa o opinião dos participantes do experimento quanto ao ambiente integrado e o não integrado, visando avaliar qual ambiente melhor auxilia o engenheiro de software na identificação dos indícios de ITs. Essa avaliação foi realizada por meio de um questionário entregue aos participantes logo após o término do experimento.

6.3.1.5 Perspectiva

O experimento foi realizado sob a perspectiva do engenheiro de software que deseja encontrar indícios de ITs em um software.

6.3.1.6 Contexto

O experimento foi realizado por alunos de graduação em Ciência da Computação da Universidade Federal de São Carlos, que concordaram em participar do experimento ao assinarem um termo de consentimento análogo ao do Apêndice C.

6.3.2 Planejamento

Esse experimento foi planejado de forma a responder a quatro questões de pesquisa: Q1: "A utilização do ambiente integrado diminui a quantidade de tempo necessária para realizar a análise do sistema?"; Q2: "A utilização do ambiente integrado permite que mais interesses transversais sejam identificados?"; Q3: "A utilização do ambiente integrado diminui a ocorrência de falsos negativos?"; Q4: "A utilização do ambiente integrado diminui a ocorrência de falsos positivos?".

Todos os participantes realizaram a análise de dois sistemas diferentes, porém com complexidade semelhante, utilizando o ambiente integrado e o não integrado. Para responder a primeira questão foi medida a quantidade de tempo utilizada para realizar a análise. Para responder as três últimas questões foram coletados os resultados com os indícios de ITs identificados pelos participantes, de forma a serem quantificados os indícios, os falsos negativos e os falsos positivos presentes nesse resultado.

6.3.2.1 Seleção do contexto

O experimento foi realizado com trinta alunos de graduação, na disciplina de "Metodologias de Desenvolvimento de Software 1", em Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos (UFSCar), no segundo semestre de 2013. Todos os participantes possuíam conhecimento prévio de Orientação a Objetos e da linguagem de programação Java.

6.3.2.2 Formulação das Hipóteses

A questões de pesquisa foram formuladas como segue:

- Q1, Hipótese Nula (H_0): não há diferença significativa entre o tempo total necessário para a identificação dos indícios de ITs utilizando o ambiente integrado (AI) e o não integrado (ANI);

$$H_0 : \mu_{AI} = \mu_{ANI}$$

- Q1, Hipótese Alternativa (H_1): o tempo total necessário para identificar os indícios de ITs é significantemente menor quanto utilizado o ambiente integrado;

$$H_1 : \mu_{AI} < \mu_{ANI}$$

- Q2, Hipótese Nula (H_0): não há diferença significativa entre a quantidade de indícios de ITs encontrada utilizando-se o ambiente integrado e o não integrado;

$$H_0 : \mu_{AI} = \mu_{ANI}$$

- Q2, Hipótese Alternativa (H_1): a quantidade de indícios encontrados é significantemente maior quando utilizado o ambiente integrado para a realização da análise;

$$H_1 : \mu_{AI} < \mu_{ANI}$$

- Q3, Hipótese Nula (H_0): não há diferença significativa entre a quantidade de falsos negativos existente nos resultados das análises por meio da ambiente integrado e do não integrado;

$$H_0 : \mu_{AI} = \mu_{ANI}$$

- Q3, Hipótese Alternativa (H_1): a quantidade de falsos negativos é significantemente menor quando utilizado o ambiente integrado para a realização da análise;

$$H_1 : \mu_{AI} < \mu_{ANI}$$

- Q4, Hipótese Nula (H_0): não há diferença significativa entre a quantidade de falsos positivos existente nos resultados das análises realizadas por meio do ambiente integrado e do não integrado;

$$H_0 : \mu_{AI} = \mu_{ANI}$$

- Q4, Hipótese Alternativa (H_1): a quantidade de falsos positivos é significantemente menor quando utilizado o ambiente integrado para a realização da análise.

$$H_1 : \mu_{AI} < \mu_{ANI}$$

6.3.2.3 Seleção das Variáveis

As variáveis dependentes consideradas neste experimento foram: "tempo necessário para realizar a análise", "número de indícios de ITs encontrados", "número de falsos negativos existente no resultado na análise" e "número de falsos positivos existente no resultado da análise". As variáveis independentes consideradas foram:

- **Aplicações:** todo participante teve que analisar os indícios dos ITs do software GrenJ (DURELLI, 2008), um *framework* do domínio de transações de aluguel, comercialização e manutenção de bens e serviços, e de um software, aqui chamado de Veterinária, do domínio de uma veterinária desenvolvido utilizando a ferramenta F3T (VIANA *et al.*, 2013). Como o GrenJ possui aproximadamente 70 classes e 18 mil linhas de código, os participantes analisaram uma versão adaptada, a mesma utilizada na prova de conceito do Capítulo 5.
- **Ferramentas utilizadas para análise:** ConcernMapper, ComSCId, FINT, XScan e SourceMiner (CARNEIRO *et al.*, 2010) (com as alterações explicadas no Capítulo 5).
- **Ambiente de Desenvolvimento:** Eclipse 3.6.2 (Eclipse Foundation, 2013).
- **Tecnologias:** Java version 7.

6.3.2.4 Seleção dos Participantes

Os participantes foram selecionados por meio de uma amostragem não probabilística e por conveniência, ou seja, não é conhecida a probabilidade correspondente dos elementos de toda a população em relação a essa amostra.

6.3.2.5 Projeto do Experimento

Os participantes foram divididos em dois grupos de quinze participantes:

- **Grupo 1:** realizou a análise do software GrenJ com o ambiente não integrado e a análise do software Veterinária com o ambiente integrado;
- **Grupo 2:** realizou a análise do software GrenJ com o ambiente integrado e a análise do software Veterinária com o ambiente não integrado.

Optou-se por dividir os participantes em grupos para reduzir a interferência causada pela experiência dos participantes, que foi determinada por meio de um questionário (Apêndice D) aplicado uma semana antes do piloto do experimento. Nesse questionário os participantes responderam sobre a suas experiências quanto à Orientação a Objetos, à Orientação a Aspectos, à linguagem de programação Java, à Mineração de Interesses Transversais, à ConcernMapper, à ComSCid, à FINT, à XScan, à SourceMiner e ao ambiente de desenvolvimento Eclipse. A análise desse questionário apresentou que os participantes não possuíam nenhuma experiência quanto à MIT, à visualização de software e às ferramentas utilizadas, portanto, para a separação dos grupos foi utilizado o nível de experiência dos participantes quanto a Orientação a Objetos, a Orientação a Aspectos e o ambiente Eclipse.

6.3.2.6 Projeto dos Tipos

O experimento foi estruturado de acordo com o projeto "um fator com dois tratamentos pareados" (WOHLIN *et al.*, 2000). O fator desse experimento refere-se a maneira como o ambiente integrado foi utilizado na análise dos indícios de ITs e os tratamentos referem-se ao apoio fornecido pelo ambiente integrado comparado com o oferecido pelo não integrado.

6.3.2.7 Instrumentação

Todo o material necessário para a realização do experimento foi previamente entregue aos participantes. Esse material era composto por um formulário para coletar dados experimentais, neste caso, tempo necessário para realizar a análise. Ao final do experimento, foi aplicado um questionário a todos os participantes para que avaliassem o ambiente integrado e o não integrado.

6.3.3 Operação

A fase de operação foi dividida em duas partes, como apresentado nas próximas subseções:

6.3.3.1 Preparação

Por meio de um formulário de caracterização foi avaliado os conhecimentos dos participantes quanto à Orientação a Objetos, à Orientação a Aspectos, à linguagem de programação Java, à Mineração de Interesses Transversais, à ConcernMapper, à ComSCid, à FINT, à XScan, à SourceMiner e ao ambiente de desenvolvimento Eclipse. Depois os participantes receberam treinamento sobre a Orientação a Aspectos, a MIT, a visualização de software e as ferramentas de MIT e de visualização de software a serem utilizadas no experimento.

6.3.3.2 Execução

Primeiramente, os participantes assinaram o formulário de consentimento. Em seguida receberam um treinamento quanto à Orientação a Aspectos, a MIT e a visualização de software. No treinamento sobre MIT foram apresentadas as técnicas das ferramentas de MIT utilizadas no experimento, bem como suas principais características e seu funcionamento. No treinamento sobre visualização de software foram apresentadas as principais características dessa área. Como ferramenta de visualização foi apresentada a SourceMiner, sendo apresentadas as suas visões e o seu funcionamento. Por fim foi apresentado como utilizar o ambiente integrado das ferramentas de MIT com a SourceMiner.

O piloto do experimento foi executado logo após a finalização do treinamento. Para tanto, os participantes foram divididos em dois grupos de acordo com suas respostas no formulário de caracterização. O piloto tinha como objetivo representar um experimento real, para que dúvidas quanto aos passos a serem realizados e o preenchimento dos formulários não influenciassem na tomada de tempo durante o experimento. Antes da execução do piloto, somente foi explicado como preencher o formulário de coleta de dados, não sendo dada nenhuma informação sobre os softwares a serem analisados.

Tendo os participantes entendido o formulário de coleta de dados e preparado seus ambiente, foi iniciada a tomada de tempo para a análise dos sistemas. Todos os participantes tiveram que analisar dois sistemas, diferentes dos utilizados no experimento, porém com grau de dificuldade similar, utilizando tanto o ambiente integrado, quanto o não integrado.

6.3.4 Análise dos Dados

Esta seção apresenta os resultados estatísticos dos dados coletados do experimento. A análise está dividida em duas subseções: Estatística Descritiva (Seção 6.3.4.2) e Teste de Hipótese (Seção 6.3.4.3).

O resultado da análise dos participantes foi coletado por meio do arquivo IMM gerado pela ferramenta ComSCId. Essa ferramenta foi escolhida em detrimento das demais, pois: i) o resultado da XScan não pode ser refinado; ii) os resultados da FINT não são separados por ITs e; iii) a identificação manual de todos os indícios por meio da ConcernMapper iria afetar a tomada de tempo.

Antes de apresentar os dados coletados é necessário apresentar dois softwares desenvolvidos durante a quantificação dos indícios, dos falsos negativos e dos falsos positivos presentes nos arquivos IMM gerados pelos participantes. Esses softwares chamam-se *Processer* e *Splitter*, e são definidas na Seção 6.3.4.1.

6.3.4.1 *Processer* e *Splitter*

Durante a análise dos dados coletados decidiu-se desenvolver um software para calcular a quantidade de indícios, de falsos positivos e de falsos negativos presentes no arquivo IMM entregue pelos participantes. Essa ferramenta chama-se *Processer* e é capaz de dado um arquivo de oráculo quantificar os indícios, os falsos positivos e os falsos negativos presentes em um arquivo IMM. Um arquivo de oráculo possui todos os indícios de um IT presentes em um software e foi representado utilizando o formato do IMM.

O *Processer* realiza a comparação das estruturas afetadas presentes em diversos arquivos IMM com as estruturas afetadas presentes no arquivo de oráculo. Essa comparação é realizada por meio dos `IDs` dessas estruturas, que pode variar de acordo com a ferramenta utilizada. Como foi solicitado no experimento que todos os indícios identificados deveriam estar presentes no arquivo IMM da ComSCId, ao gerar um oráculo utilizando essa ferramenta, a comparação por `IDs` não gerou um problema.

Como o nome do IT procurado foi definido pelo participante, decidiu-se que a *Processer* somente executaria a análise de IT por vez, sendo que todos os indícios presentes no IMM analisado devem pertencer a um mesmo IT. Para garantir essa premissa, necessitou-se o desenvolvimento de outro software, chamado *Splitter*. Esse software analisa um arquivo IMM e separa os indícios de cada um dos ITs em seu um novo arquivo.

Os resultados da comparação realizada pela *Processer* são exibidos em um arquivo texto, separados por IMM analisado e contém a somatória total de: i) indícios presentes tanto no IMM do participante quanto no do oráculo; ii) indícios presentes somente no IMM do oráculo (falsos negativos) e iii) indícios presentes somente no IMM do participante (falsos positivos). Para possibilitar a avaliação dos falsos positivos e negativos, juntamente com sua somatória é listado o `ID` de todas as estruturas que compõe esse resultado.

De posse dessas informações faltou somente calcular o tempo total gasto pelo participante, que foi realizado por meio do formulário de coleta de dados. Esses dados, bem como suas análises são apresentados nas próximas seções.

6.3.4.2 Estatísticas Descritivas

Ao analisar os dados dos participantes, percebeu-se que alguns deles não realizaram a identificação dos indícios do IT de persistência como solicitados, gerando somente o arquivo da ComSCId sem a alteração de nenhuma de suas regras. Esses participantes foram removidos de forma a não interferir no resultado dos demais. Para equilibrar os grupos, alguns participantes foram aleatoriamente removidos. Com isso o total de participantes analisados foi de vinte e dois, sendo oito participantes removidos. Esses dados são apresentados na Tabela 6.2. Os dados coletados foram separados entre ambiente integrado (AI) e ambiente não integrado (ANI).

Primeiramente deve-se verificar a existência de *outliers*, que apresentam um comportamento diferente dos demais. Para isso foi utilizado o gráfico de *boxplot*, para representar os dados de cada uma das colunas da Tabela 6.2. Esses gráficos são apresentados na Figura 6.4.

Como apresentado nas Figuras 6.4(d), 6.4(f), 6.4(g) e 6.4(d) os participantes P1, P2, P3, P11, P12, P16, P17 e P20 são *outliers* e, portanto, devem ser removidos da análise de inferência das hipóteses. A nova tabela de dados é apresentada na Tabela 6.3.

O próximo passo é determinar se esses dados provêm ou não de uma distribuição normal. Essa informação é necessária para decidir o teste de inferência a ser aplicado para avaliar as hipóteses. Para essa análise foi utilizado o teste de Shapiro-Wilk, aplicado com o auxílio do software Action (ESTATCAMP, 2013e). Todas as colunas da Tabela 6.3 foram analisadas, uma a uma, e os resultados podem ser vistos nas Figuras 6.6. Como todos os P-valores apresentados são menores que $\alpha = 0,05$, pode-se determinar, com um nível de significância de 95% que os dados não provêm de uma distribuição normal. Para inferência das hipóteses deve-se então aplicar o teste de Wilcoxon[3].

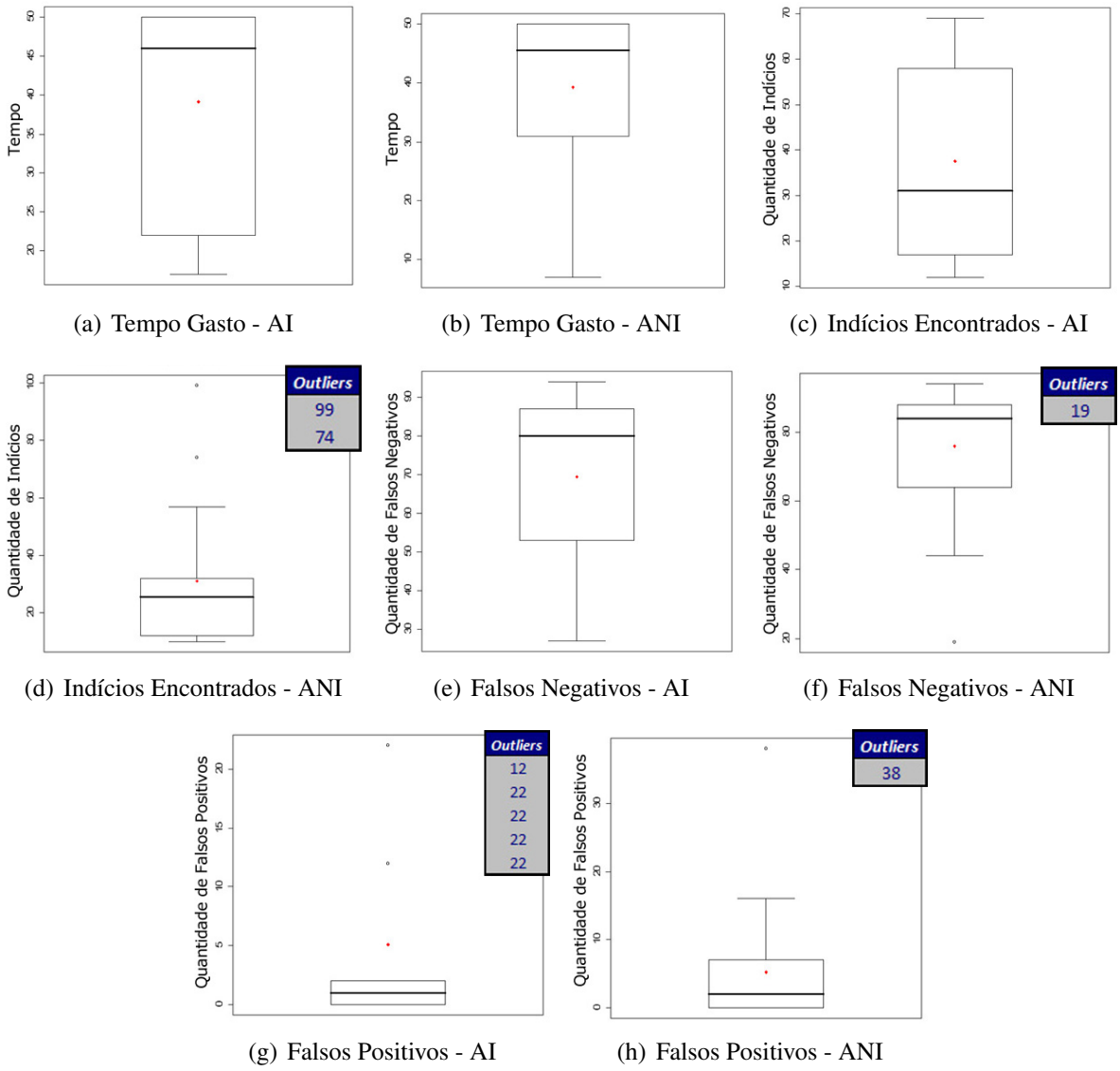
[3]http://www.portalaction.com.br/974-3-teste-de-wilcoxon-pareado

Tabela 6.2: Dados coletados no experimento de utilização do ambiente integrado.

	Tempo Gasto		Indícios Encontrados		Falsos Negativos		Falsos Positivos	
Participante	AI	ANI	AI	ANI	AI	ANI	AI	ANI
P1	50	46	69	99	27	19	0	2
P2	17	7	16	30	80	88	0	38
P3	50	50	17	48	79	70	12	7
P4	50	50	12	57	84	61	0	2
P5	19	50	16	24	80	94	0	2
P6	42	50	58	57	38	61	0	2
P7	22	50	16	24	80	94	0	2
P8	22	22	16	24	80	94	0	2
P9	22	22	58	24	38	94	0	2
P10	22	17	59	27	37	91	0	2
P11	47	50	43	74	53	44	0	2
P12	50	35	69	32	49	64	22	16
P13	50	22	31	32	87	64	2	12
P14	50	45	57	12	61	84	2	0
P15	42	50	24	12	94	84	1	0
P16	50	50	25	10	93	86	22	12
P17	50	31	59	27	59	69	22	0
P18	20	36	31	12	87	84	2	0
P19	50	50	24	10	94	86	2	0
P20	49	49	59	12	59	84	22	0
P21	42	37	36	10	82	86	1	0
P22	45	45	31	27	87	69	2	12
Média	39.1364	39.2727	37.545455	31.090909	69.4545	75.90909	5.09091	5.227273
%	49.913	50.087	54.701987	45.298013	47.7799	52.22014	49.3392	50.66079

Tabela 6.3: Dados coletados no experimento de utilização do ambiente integrado após retirada dos *outliers*.

	Tempo Gasto		Indícios Encontrados		Falsos Negativos		Falsos Positivos	
Participante	AI	ANI	AI	ANI	AI	ANI	AI	ANI
P4	50	50	12	57	84	61	0	2
P5	19	50	16	24	80	94	0	2
P6	42	50	58	57	38	61	0	2
P7	22	50	16	24	80	94	0	2
P8	22	22	16	24	80	94	0	2
P9	22	22	58	24	38	94	0	2
P10	22	17	59	27	37	91	0	2
P13	50	22	31	32	87	64	2	12
P14	50	45	57	12	61	84	2	0
P15	42	50	24	12	94	84	1	0
P18	20	36	31	12	87	84	2	0
P19	50	50	24	10	94	86	2	0
P21	42	37	36	10	82	86	1	0
P22	45	45	31	27	87	69	2	12
Média	35.5714	39	33.5	25.142857	73.5	81.85714	0.85714	2.714286
%	47.7011	52.2989	57.125457	42.874543	47.3103	52.68966	24	76



(a) Tempo Gasto - AI (b) Tempo Gasto - ANI (c) Indícios Encontrados - AI

(d) Indícios Encontrados - ANI (e) Falsos Negativos - AI (f) Falsos Negativos - ANI

(g) Falsos Positivos - AI (h) Falsos Positivos - ANI

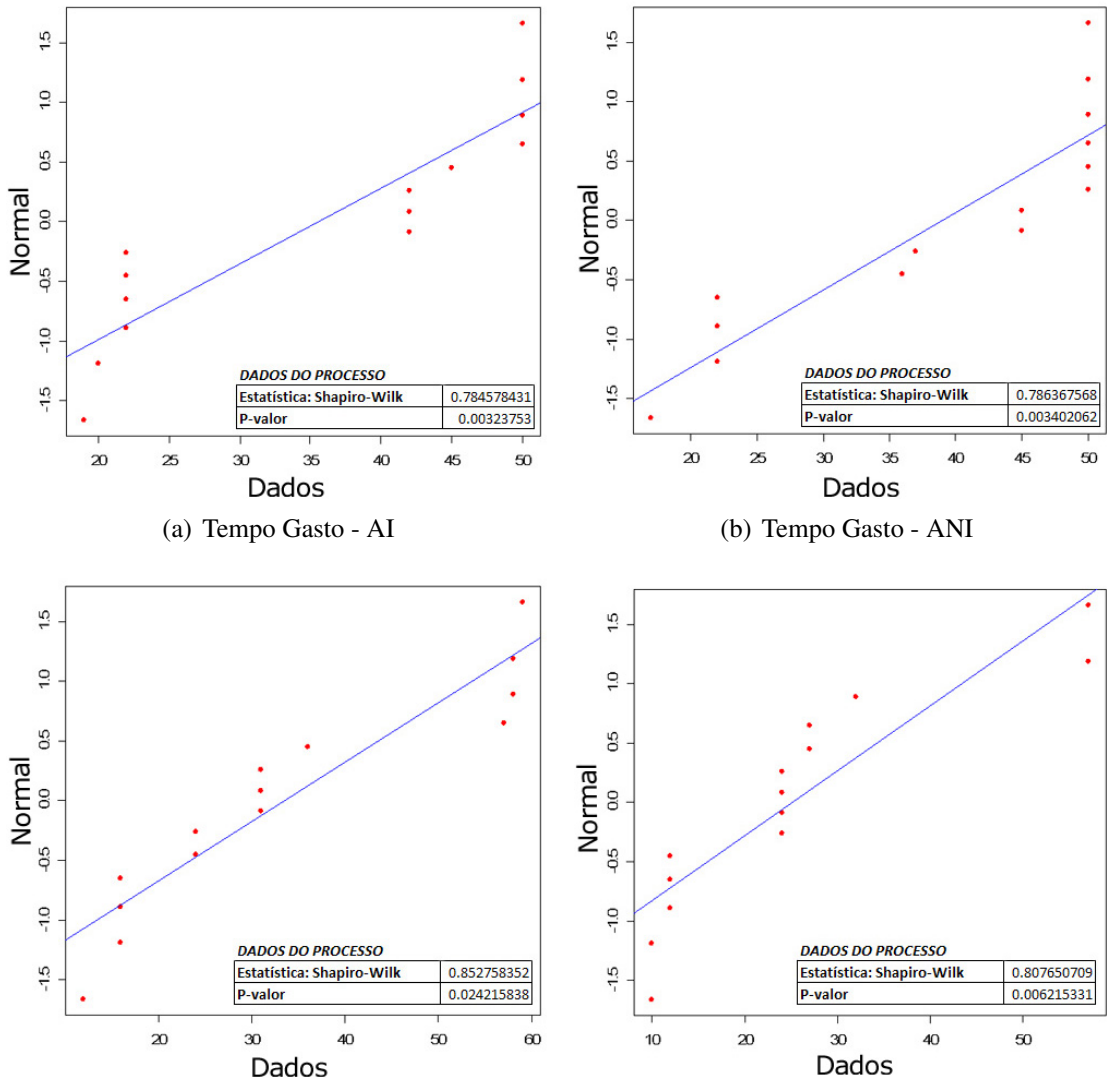
Figura 6.4: Boxplots dos dados da Tabela 6.2.

6.3.4.3 Testando as Hipóteses

Para inferência das hipóteses será usado o teste de Wilcoxon Pareado, com o auxílio do software Action, a fim de avaliar as hipóteses das quatro questões de pesquisa. Os resultados são apresentados na Figura 6.7.

Para o tempo, apresentado na Figura 6.7(a), P-valor é aproximadamente $0,15$, que é maior do que $\alpha = 0,05$. Portanto, para um nível de significância de 95%, não se pode rejeitar a hipótese nula, que é: "não há diferença significativa entre o tempo total necessário para a identificação dos indícios de ITs utilizando o ambiente integrado e o não integrado".

Para os indícios encontrados, apresentado na Figura 6.7(b), P-valor é aproximadamente $0,92$, que é maior que $\alpha = 0,05$. Portanto, para um nível de significância de 95%, não se pode



(a) Tempo Gasto - AI (b) Tempo Gasto - ANI

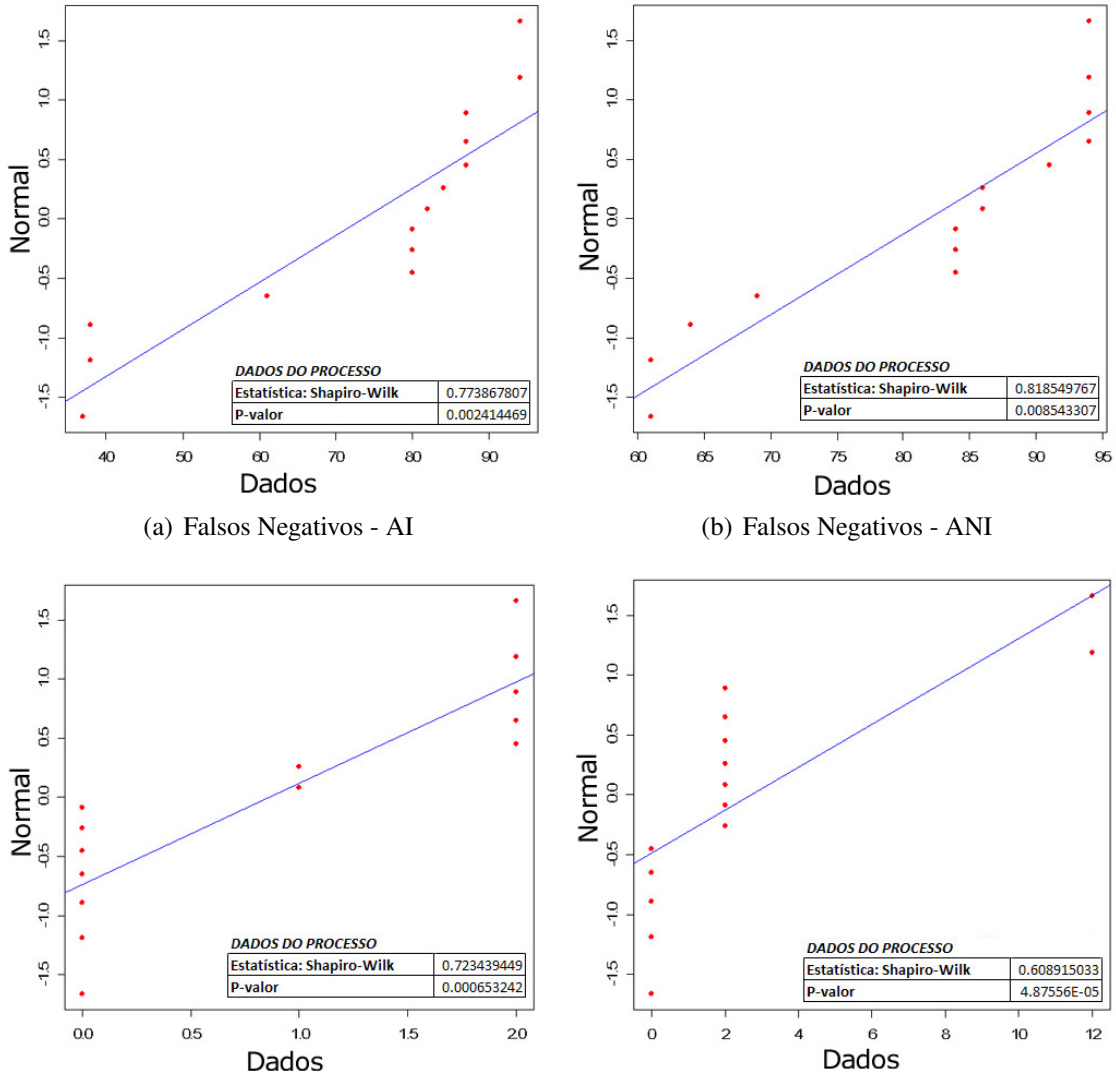
(c) Indícios Encontrados - AI (d) Indícios Encontrados - ANI

Figura 6.5: Testes de normalidade dos dados da Tabela 6.3. Parte 1.

rejeitar a hipótese nula, que é: "não há diferença significativa entre a quantidade de indícios de ITs encontrada utilizando-se o ambiente integrado e o não integrado".

Para os falsos negativos, apresentado na Figura 6.7(c), P-valor é aproximadamente $0,17$, que é maior que $\alpha = 0,05$. Portanto, para um nível de significância de 95%, não se pode rejeitar a hipótese nula, que é: "não há diferença significativa entre a quantidade de falsos negativos presente nos resultados das análises por meio da ambiente integrado e do não integrado".

Para os falsos positivos, apresentado na Figura 6.7(d), P-valor é aproximadamente $0,04$, que é menor que $\alpha = 0,05$. Portanto, para um nível de significância de 95%, pode-se rejeitar a hipótese nula, aceitando-se a hipótese alternativa que é: "a quantidade de falsos positivos é significantemente menor quando utilizado o ambiente integrado para a realização da análise".



(a) Falsos Negativos - AI

(b) Falsos Negativos - ANI

(c) Falsos Positivos - AI

(d) Falsos Positivos - ANI

Figura 6.6: Testes de normalidade dos dados da Tabela 6.3. Parte 2.

Com isso, somente a última hipótese nula foi refutada. Para que se possa apresentar uma interpretação para esse resultado, precisa-se primeiro avaliar a opinião dos participantes, apresentada na próxima seção.

6.3.5 Ameaças à Validade

É importante considerar o quão válidos os resultados são, sendo assim, esta seção apresenta algumas das ameaças que podem afetar o valores coletados e a conclusão obtida.

1. Validade interna

   - O nível de conhecimento dos participantes era bastante variado. Visando ame-

Informação	Valor
V	13.5
P-valor	0.155688889
Hipótese Nula	0
Método	Wilcoxon signed rank test with continuity correction
(Pseudo) Mediana	-5.499947377
Intervalo de Confiança - Limite Superior	5.000003767

(a) Wilcoxon Pareado - Tempo

Informação	Valor
V	75
P-valor	0.925911064
Hipótese Nula	0
Método	Wilcoxon signed rank test with continuity correction
(Pseudo) Mediana	9.000021416
Intervalo de Confiança - Limite Superior	18.99995722

(b) Wilcoxon Pareado - Indícios Encontrados

Informação	Valor
V	37
P-valor	0.172352111
Hipótese Nula	0
Método	Wilcoxon signed rank test with continuity correction
(Pseudo) Mediana	-6.17960293
Intervalo de Confiança - Limite Superior	4.500058653

(c) Wilcoxon Pareado - Falsos Negativos

Informação	Valor
V	25.5
P-valor	0.041234323
Hipótese Nula	0
Método	Wilcoxon signed rank test with continuity correction
(Pseudo) Mediana	-1.999975596
Intervalo de Confiança - Limite Superior	-3.03949E-05

(d) Wilcoxon Pareado - Falsos Positivos

Figura 6.7: Testes de Inferência.

nizar essa ameaça os participantes foram divididos em dois grupos equilibrados, considerando-se o nível de conhecimento dos participantes. Os participantes também receberam treinamento prévio sobre as ferramentas utilizadas e como utilizar o ambiente integrado e o não integrado para a detecção dos indícios de ITs. Mesmo com o treinamento, durante a execução do experimento, percebeu-se que os participantes apresentaram muitas dificuldades para utilizar os ambientes integrado e não integrado. Acredita-se que essa dificuldade influenciou os resultados;

- Computadores e instalações diferentes podem afetar o resultado. Visando amenizar essa ameaça, o experimento foi realizado em Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos (UFSCar);
- O receio dos participantes de que seriam avaliados de acordo com os resultados do experimento. Visando amenizar essa ameaça, foi claramente explicado que o experimento não seria, de forma alguma, utilizado para avaliá-los.

2. Validação pela Construção

   - Os participantes conheciam os pesquisadores e sabiam que a utilização do ambiente integrado poderia ser mais rápido, o que poderia influenciar os resultados, tornando o experimento menos imparcial. Para amenizar essa ameaça, conduziu-se o experimento de forma que os participantes mantivessem um ritmo constante.

3. Validade Externa

- É possível que as atividades propostas durante o experimento não representem situações do mundo real. Para amenizar essa ameaça, foram selecionados sistemas cujos códigos fonte estão abertos.

4. Validade da Conclusão

   - Os resultados obtidos são influenciados diretamente pela precisão das métricas utilizadas durante a execução do experimento. Visando amenizar essa ameaça, utilizou-se somente o tempo total gasto pelos participantes para encontrar os indícios de ITs e a assertividade do resultado obtido;
   - A quantidade de participantes foi pequena, o que apresenta um baixo poder estatístico. Visando suavizar essa ameaça, aplicou-se o de Wilcoxon Pareado.

6.3.6 Opinião dos participantes quanto à utilização do ambiente integrado de ferramentas de MIT e de visualização de software

Um questionário (Figura 6.8), foi elaborado e aplicado aos participantes do experimento para avaliar a opinião dos participantes quanto à utilização do ambiente integrado de ferramentas de MIT e de visualização de software. Esse questionário foi aplicado após a execução do experimento, de modo que as respostas forneçam subsídios para a comparação do uso do ambiente integrado e do não integrado, bem como uma estimativa que representasse a quantidade de auxílio proporcionado por cada ferramenta utilizada.

Responderam esse questionário os trinta participantes do experimento. As respostas foram sumarizadas e são apresentadas na Figura 6.9. A primeira coluna do gráfico expressa a indicação dos participantes em 70% para o auxílio oferecido pelo ambiente integrado e de 55% para quando não foi usada a integração. 75% dos participantes utilizariam o ambiente integrado em futuros processos de identificação de indícios de ITs, enquanto que 40% utilizariam o ambiente não integrado.

Um dos problemas detectados foi a falta de conhecimento dos participantes, como comentado anteriormente e afirmado pela maioria, 80%. A necessidade de conhecimento das ferramentas do ambiente integrado para a realização da análise, apresentou resultado considerável e é expresso na Figura 6.9 pelas colunas de 7 a 11. Pode-se perceber que a necessidade de conhecimento das ferramentas ComSCId e SourceMiner é de 85%, enquanto das demais é de 50 a 55%.

A utilização do ambiente integração não obriga o participante a usar todas as ferramentas. Como premissa do experimento, o participante deveria agrupar todos os indícios encontrados

AVALIAÇÃO DO AMBIENTE INTEGRADO DE MINERAÇÃO DE INTERESSES TRANSVERSAIS E VISUALIZAÇÃO DE SOFTWARE

Em sua opinião, considerando o conhecimento adquirido em aula, dê um valor para cada uma das questões abaixo.
O valor deve ser um inteiro entre 0 (zero) e 10 (dez), onde 0 significa 0% (zero por cento) e 10 significa 100% (cem por cento).

1) Qual porcentagem expressa o auxílio da abordagem integrada na identificação de Interesses Transversais (IT)? _____

2) Qual porcentagem expressa o auxílio da abordagem não integrada na identificação de ITs? _____

3) Qual a probabilidade de você utilizar a abordagem integrada para a identificação de ITs? _____

4) Qual a probabilidade de você utilizar a abordagem não integrada para a identificação dos ITs? _____

5) Qual porcentagem expressa a facilidade de utilização da abordagem integrada? _____

6) Qual porcentagem expressa a facilidade de utilização da abordagem não integrada? _____

7) Qual porcentagem expressa a necessidade de entendimento da ferramenta ComSCId para a utilização da abordagem integrada? _____

8) Qual porcentagem expressa a necessidade de entendimento da ferramenta FINT para a utilização da abordagem integrada? _____

9) Qual porcentagem expressa a necessidade de entendimento da ferramenta XScan para a utilização da abordagem integrada? _____

10) Qual porcentagem expressa a necessidade de entendimento da ferramenta ConcernMapper para a utilização da abordagem integrada? _____

11) Qual porcentagem expressa a necessidade de entendimento da ferramenta SourceMinner para a utilização da abordagem integrada? _____

12) Qual porcentagem expressa a interferência causada pelo não conhecimento das ferramentas na identificação dos ITs? _____

13) Qual porcentagem expressa o auxílio fornecido pela integração na comparação dos resultados obtidos pelas ferramentas de mineração de ITs? _____

14) Qual porcentagem expressa o auxílio fornecido pela integração na identificação de falsos positivos? _____

15) Qual porcentagem expressa o auxílio fornecido pela integração na identificação de falsos negativos? _____

Figura 6.8: Questionário de avaliação do ambiente integrado.

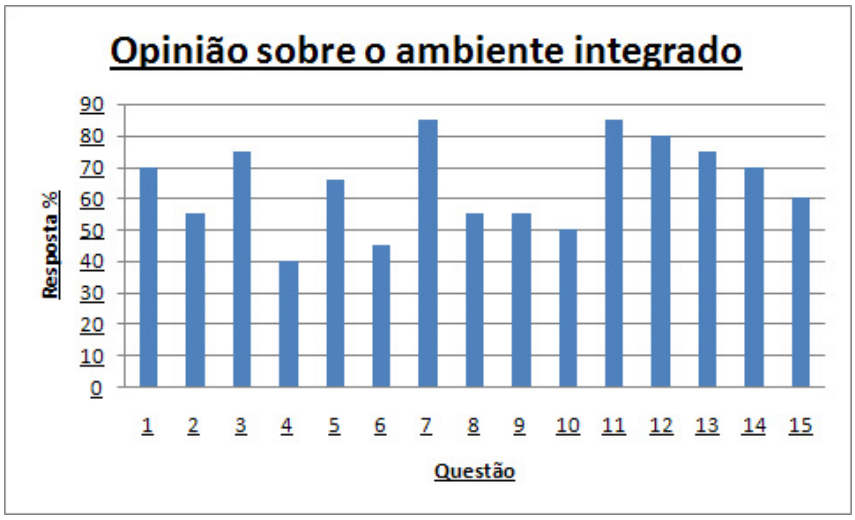


Figura 6.9: Gráfico com a opinião dos participantes.

com o apoio da ComSCId e visualizá-los na SourceMiner. Ao analisar os valores citados no parágrafo anterior, percebeu-se que a falta de conhecimento dos participantes interferiu na quantidade de ferramentas utilizadas, sendo que, em boa parte dos casos, somente foram utilizadas

as solicitadas por este autor, o que mostra que os participantes não usufruíram das vantagens da integração. Dessa forma acredita-se que a integração não foi utilizada por completo e por isso não houve diferença significativa entre a utilização do ambiente integrado e do não integrado, sendo que somente a diferença entre o número de falsos positivos foi significativa.

6.3.7 Avaliação da Usabilidade das Visualizações da SourceMiner

A partir da experiência adquirida pelos participantes na utilização da SourceMiner foi proposta uma atividade para o uso da versão proposta por Silva A. N.; Carneiro (2012), denominada, nesta avaliação, VB. O objetivo foi a avaliação, por meio de um questionário (Figura 6.11), das visões apresentadas pelas duas versões da ferramenta. Com a aplicação do questionário foi verificada a usabilidade da ferramenta nas duas versões e qual a interface mais amigável para o usuário, bem como avaliar as alterações realizadas na SourceMiner proposta por Carneiro *et al.* (2010), denominada, nesta avaliação, VA.

Somente dez participantes do experimento realizaram a análise, pois não era uma atividade obrigatória. A compilação das respostas está representada no gráfico apresentado na Figura 6.10.

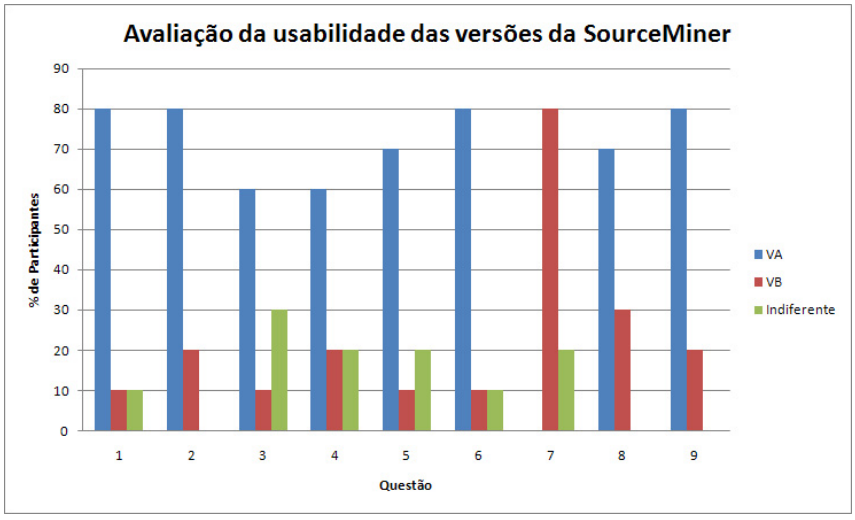


Figura 6.10: Resultados do questionário de usabilidade das versões da SourceMiner.

Como resultado dessa atividade nota-se que a VA é a que apresenta a interface mais amigável. Vale destacar a resposta da questão 7, que indica que a falta de *scrollbar* prejudicou bastante a utilização da VB com o ambiente integrado.

QUESTIONÁRIO INDIVIDUAL PARA AVALIAÇÃO DA USABILIDADE DAS VISUALIZAÇÕES DE DUAS VERSÕES DA SOURCEMINER

Data:

Para responder as perguntas abaixo considere:
VA – versão da SourceMiner utilizada nos experimentos
VB – versão da SourceMiner utilizada na tarefa

1) Qual versão apresenta maior facilidade para exibição de suas abas, e conseqüente utilização?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

2) Qual versão é mais efetiva para a visualização dos indícios de interesses transversais?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

3) Qual versão é mais efetiva para a visualização dos indícios de interesses transversais em seus respectivos pacotes e classes?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

4) Considerando a aba *ConcernFilterView*, qual versão é mais efetiva para colorir os indícios de interesses transversais?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

5) Considerando a aba *ConcernFilterView*, qual versão apresenta melhor disposição dos seus elementos (*color picker*, nome do interesse e *check box*) ?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

6) Considerando a aba ConcernFilterView com grande quantidade de indícios, qual versão é mais efetiva para a visualização e coloração dos indícios nessa aba?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

7) Considerando a aba ConcernFilterView com grande quantidade de indícios, qual versão impossibilitou a visualização e coloração e alguns dos indícios encontrados?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

8) Qual versão possui a interface com maior usabilidade (amigável, auto explicativa, coloração)?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

9) Em sua opinião, qual versão é mais efetiva para ser utilizada em conjunto com as ferramentas de mineração?
( ) VA ( ) VB ( ) Indiferente
Justifique:_____
_____
_____

Figura 6.11: Questionário de usabilidade das versões da SourceMiner.

6.4 Considerações Finais

O teste qualitativo do segundo experimento reforçou a premissa de que o pouco conhecimento dos participantes sobre as ferramentas integradas interferiu na utilização do ambiente integrado, sendo utilizadas principalmente as duas ferramentas solicitadas, ComSCId e SourceMiner. Um dos pontos positivos da integração é a comparação dos resultados obtidos pelas diversas ferramentas de MIT. Como na maior parte do experimento os participantes utilizaram somente uma ferramenta de MIT, não houve diferença significativa entre a utilização do ambiente integrado e do não integrado. Focando-se somente nos falsos positivos pode-se perceber que os novos padrões visuais apresentados ao exibir os indícios encontrados pela ComSCId

nas visões da SourceMiner permitiu uma redução significativa no número de falsos positivos, indicando que a integração auxilia na identificação dos falsos positivos apresentados por uma ferramenta.

Com isso concluiu-se que, apesar do IMM apresentar indícios de que facilita a integração das ferramentas de MIT e de visualização de software, como apresentado pelo resultado do primeiro experimento, ainda é necessário que o engenheiro de software domine cada uma das ferramentas integradas para que possa aproveitar todos os benefícios propostos pelo ambiente integrado. Dessa forma, para que se possa avaliar esse ambiente e obter-se um resultado expressivo é necessário que os participantes tenham forte conhecimento prévio das ferramentas de MIT e de visualização de software, ou que seja possível utilizar um longo período de tempo para treinar os participantes na utilização dessas ferramentas.

Por fim, o questionário de usabilidade das versões da SourceMiner aponta que as visões da primeira versão são mais amigáveis ao usuário no contexto do ambiente integrado e que as alterações realizadas na SourceMiner, identificadas durante sua utilização neste trabalho, também são pertinentes.

Capítulo 7

CONCLUSÕES

Um software nem sempre apresenta modularização adequada, de forma que sua manutenção nem sempre ocorre de forma trivial. Os ITs estão espalhados e entrelaçados em seu código fonte, sendo que técnicas de mineração de interesses transversais (MIT) (ZHANG; JACOBSEN, 2012) devem ser usadas para que a modularização possa ser obtida. Se esse processo for feito manualmente, várias dificuldades são encontradas, tais como: deficiência na identificação desses ITs, o alto custo e longo tempo de identificação e os erros que podem ser inseridos no código fonte. O engenheiro de software também não tem garantia que o uso de uma única técnica de mineração possa encontrar todos os indícios de ITs existentes no código fonte. Como trabalhos já identificaram (CECCATO *et al.*, 2006) é interessante o uso combinado de técnicas de mineração para que o maior número de ITs possa ser identificado.

Uma das formas de minimizar os problemas elencados anteriormente é com o uso das ferramentas de MIT (NGUYEN *et al.*, 2011; PARREIRA JÚNIOR *et al.*, 2010a; ROBILLARD; WEIGAND-WARR, 2005; MARIN *et al.*, 2007), que possuem enfoque na identificação dos indícios de ITs que afetam um sistema. Algumas vezes essas ferramentas não apresentam seus resultados de forma satisfatória, por não utilizarem recursos gráficos para exibi-los. Uma forma interessante de melhorar a visualização desses resultados é apresentá-los por meio das visões de ferramentas de visualização de software (DELFIM; GARCIA, 2013; BENTRAD; MESLATI, 2011; CARNEIRO *et al.*, 2010). Entretanto, a integração de ferramentas de MIT e de visualização software nem sempre ocorrem sem um árduo trabalho de alteração de código fonte, pois, normalmente, os padrões utilizados pelas ferramentas MIT para referenciar as estruturas afetadas são diferentes, com particularidades e pouco documentados.

A motivação para realização deste trabalho surgiu a partir do contexto anteriormente apresentado e por não ter sido encontrado uma forma facilitada para integração dessas ferramentas. Assim, o objetivo desta dissertação foi possibilitar a utilização integrada de diversas ferramentas

de MIT com ferramentas de visualização. Para isso foi criado e utilizado o metamodelo *Indications Metamodel* (IMM), capaz de identificar os indícios de ITs do código fonte, de modo a possibilitar a integração das ferramentas de MIT com as de visualização de software.

Essa integração possibilita que o engenheiro de software encontre e visualize os ITs existentes em um software, para posteriormente refiná-los. A integração ou combinação dessas ferramentas de MIT facilita a comparação de seus resultados. Dessa forma, os falsos negativos são evidenciados (CECCATO *et al.*, 2006) e o resultado é complementado, apoiando o engenheiro de software na melhoria do um software, uma vez que os resultados são apresentados com formas gráficas e técnicas de MIT distintas. As visões das ferramentas de visualização também proporcionam a identificação de novos padrões visuais, que podem ser utilizados para identificar métodos que possuem maior probabilidade de indicarem um falso positivo. Com isso a análise do software é facilitada e permite que o resultado final seja melhorado, combinando os pontos positivos das ferramentas de MIT com os das ferramentas de visualização de software.

Nas Seções 7.1 e 7.2 são descritas, respectivamente, as contribuições e limitações deste trabalho. Na Seção 7.3 são apresentadas sugestões de trabalhos futuros.

7.1 Contribuições deste Trabalho

O levantamento bibliográfico realizado compilou um conjunto de características existentes em cada uma das ferramentas de mineração e de visualização. Esse resultado contribui para que os pesquisadores possam conhecer detalhadamente cada uma das ferramentas no desenvolvimento de software.

Outra contribuição deste trabalho foi a construção do metamodelo IMM como uma forma de identificar os indícios de ITs encontrados no código fonte, servindo como meio de comunicação entre as ferramentas de MIT e de visualização de software analisadas. Sua utilização facilita a integração dessas ferramentas, como pôde ser visto nos resultados relatados no Capítulo 6, referente ao experimento realizado com a integração da ferramenta ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a) com a SourceMiner (CARNEIRO *et al.*, 2010).

O IMM é genérico e possibilita que suas classes sejam estendidas, permitindo que novas classes e atributos sejam adicionadas a ele. Com isso, mesmo ferramentas de MIT e de visualização de software que possuam alguma particularidade podem ser integradas.

Neste trabalho, o IMM foi utilizado para integrar seis ferramentas: ConcernMapper (RO-

BILLARD; WEIGAND-WARR, 2005), ComSCId (PARREIRA JÚNIOR *et al.*, 2010b; PARREIRA JÚNIOR *et al.*, 2010a), FINT (MARIN *et al.*, 2007), XScan (NGUYEN *et al.*, 2011), $SoftVis_{4CA}$ (DELFIM; GARCIA, 2013; DELFIM, 2013) e SourceMiner (SILVA A. N.; CARNEIRO, 2012; CARNEIRO *et al.*, 2010). Esse ambiente integrado facilita a comparação dos resultados obtidos pelas ferramentas de MIT, visto que os resultados são apresentados de uma maneira geral nas visões das ferramentas de visualização, permitindo avaliar como um IT afeta as estruturas de um software. Essas visões também permitem a identificação de novos padrões visuais, que destacam indícios com maior probabilidade de serem falsos positivos. Com isso, a análise de um software é facilitada e o resultado final apresenta melhor qualidade, sendo que os resultados de diversas técnicas de MIT são agrupados.

Com a realização do experimento os dados obtidos apresentam que o ambiente integrado é capaz de evidenciar indícios, encontrados por uma ferramenta MIT, que apresentem maior probabilidade de serem falsos positivos. Os demais dados analisados apresentaram uma pequena diferença a favor da utilização do ambiente integrado, porém, estatisticamente, essa diferença não é significativa. Percebeu-se que, apesar do IMM facilitar a integração das ferramentas, ainda é necessário que o engenheiro de software tenha conhecimento e domínio da utilização de cada uma delas, de modo que possa se beneficiar completamente dessa integração. Como o experimento foi realizado em ambiente acadêmico, os participantes não tinham domínio das ferramentas, sendo que as mesmas foram apresentadas e por eles utilizadas de forma geral. Por fim, outro resultado apresentado pelo experimento é que o ambiente integrado é capaz de evidenciar indícios, encontrados por uma ferramenta, que apresentem maior probabilidade de serem falsos positivos.

Para quantificar a diferença entre os indícios de ITs encontrados pelas ferramentas integradas, foi desenvolvido o software *Processer*, apresentado no Capítulo 6. Esse software, por meio de um oráculo é capaz de quantificar os indícios, os falsos negativos e os falsos positivos existentes no resultado apresentado por uma ferramenta de MIT integrada. Como o *Processer* somente pode comparar um IT por vez, foi desenvolvido também um software capaz de separar, em arquivos diferentes, os indícios de cada IT presente em um arquivo IMM. Esse software chama-se *Splitter* e também foi apresentado no Capítulo 6.

Por fim, também é contribuição deste trabalho as melhorias executadas nas ferramentas ComSCId, XScan e SourceMiner, discutidas no Capítulo 5. Essas melhorias possibilitaram que a ComSCId e a XScan fossem executadas também no sistema operacional Linux. Outra melhoria executada na XScan foi a identificação dinâmica do projeto a ser analisado, evitando que seu código fonte fosse recompilado a cada novo projeto analisado. No caso da SourceMiner,

as melhorias somente foram identificadas após sua utilização integrada com as ferramentas de MIT e visavam auxiliar a usabilidade de suas visões de acordo com os novos requisitos exigidos pelo ambiente integrado. Essas alterações foram analisadas por meio de um questionário de usabilidade apresentado no Capítulo 6, que comprovou que essas alterações são pertinentes.

7.2 Limitações deste Trabalho

A linguagem Java permite que um arquivo de extensão ".java" declare mais de uma classe. A classe `Class` do IMM não armazena informação sobre o nome do arquivo Java no qual ela está implementada. Com isso tem-se uma limitação do IMM, que é a identificação de várias classes implementadas no mesmo arquivo. Existem duas formas de contornar essa limitação: i) alterar o código fonte do sistema a ser analisado para que cada classe seja implementada em seu respectivo arquivo ou; ii) decorar a classe `Class` de forma que o nome do arquivo Java seja armazenado. Isso permite a utilização dessa informação em novas ferramentas a serem integradas, porém essa modificação precisaria ser replicada nas ferramentas já utilizadas.

Durante a análise dos resultados dos experimentos, foi desenvolvido um software para calcular a quantidade de indícios encontrados, de falsos negativos e positivos. Esse software, apresentado no Capítulo 6 e chamado *Processer*, apresenta como limitações: i) a não existência de uma interface gráfica para a sua utilização; ii) a identificação das estruturas de um software é feita por meio de seus `IDs`, que podem variar de acordo com a ferramenta utilizada e; iii) somente pode analisar um IT por vez.

7.3 Sugestões de Trabalhos Futuros

Alguns trabalhos podem dar continuidade a este com o objetivo de complementá-lo e ampliá-lo de modo que o IMM possa ser utilizado para integrar mais ferramentas de MIT e de visualização de software. Nesta seção são apresentadas algumas sugestões de trabalhos futuros, que podem amenizar as limitações apresentadas na Seção 7.2.

a) **Revisão sistemática:** realizar uma revisão sistemática sobre Metamodelos para integração de ferramentas de MIT e visualização de software e analisar como esses Metamodelos podem ser reaproveitados para integrar diferentes ferramentas;

b) **Atualização do metamodelo para que o nome do arquivo Java seja armazenado:** incluir um novo atributo na classe `Class` do metamodelo de forma a armazenar o nome

do arquivo Java na qual ela está implementada. Após essa inclusão deve-se atualizar a integração das ferramentas utilizadas neste trabalho;

c) **Incluir novas ferramentas de MIT e visualização de software no ambiente integrado:** possibilitar um escopo mais amplo para a melhoria de um software com ITs espalhados e entrelaçados;

d) **Melhorias no *Processer*:** criar interface gráfica para facilitar a identificação do arquivo de oráculo e dos arquivos a serem comparados. Permitir que as estruturas do software sejam identificadas independentemente de seus `IDs`. Permitir que o *Processer* possa comparar diversos ITs simultaneamente;

e) **Criar um programa para comparação de técnicas de MIT:** após realizadas as melhorias no *Processer*, o mesmo pode ser utilizado como uma ferramenta para comparação de técnicas e ferramentas de MIT. Ao ser fornecido um software para análise, juntamente com seu arquivo de oráculo, os criadores de técnicas e ferramentas de MIT poderiam utilizá-lo para comparar suas técnicas com outras existentes na literatura. Para tornar isso possível seria necessária a criação de um manual de utilização do *Processer*, bem como a sua disponibilização em uma página para *download*;

f) **Realização de novo experimento para avaliar a utilização do ambiente integrado:** o pouco conhecimento dos participantes sobre as técnicas e ferramentas de MIT e visualização de software, interferiu no resultado do experimento. Um novo experimento pode ser realizado, porém os participantes deveriam ter um longo período para conhecimento e uso das ferramentas de MIT e de visualização, para a obtenção de um resultado expressivo.

REFERÊNCIAS

ABILIO, R.; TELES, P.; COSTA, H.; FIGUEIREDO, E. A systematic review of contemporary metrics for software maintainability. In: *Software Components Architectures and Reuse (SBCARS), 2012 Sixth Brazilian Symposium on*. [S.l.: s.n.], 2012. p. 130–139.

AMR, S. *AJHotDraw*. 2013. http://ajhotdraw.sourceforge.net/. Acesso em: outubro de 2013.

ARACIC, I.; GASIUNAS, V.; MEZINI, M.; OSTERMANN, K. An overview of caesarj. In: RASHID, A.; AKSIT, M. (Ed.). *Transactions on Aspect-Oriented Software Development I*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 3880). p. 135–173. ISBN 978-3-540-32972-5. Disponível em: <http://dx.doi.org/10.1007/11687061_5>.

ASPECTOS, V. de Software Orientado a. *Visualização de Software Orientado a Aspectos*. Dissertação (Trabalho de Conclusão de Curso) — Universidade Estadual Paulista Júlio de Mesquita Filho (UNESP), 2007.

DI BATTISTA, G.; EADES, P.; TAMASSIA, R.; TOLLIS, I. G. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 4, n. 5, p. 235–282, out. 1994. ISSN 0925-7721. Disponível em: <http://dx.doi.org/10.1016/0925-7721(94)00014-X>.

BAXTER, I.; YAHIN, A.; MOURA, L.; SANT'ANNA, M.; BIER, L. Clone detection using abstract syntax trees. In: *Software Maintenance, 1998. Proceedings., International Conference on*. [S.l.: s.n.], 1998. p. 368–377. ISSN 1063-6773.

BENTRAD, S.; MESLATI, D. 2d and 3d visualization of aspectj programs. In: *Programming and Systems (ISPS), 2011 10th International Symposium on*. [S.l.: s.n.], 2011. p. 183–190.

BERNARDI, M. L.; LUCCA, G. A. Identifying the crosscutting among concerns by methods' calls analysis. In: KIM, T.-h.; ADELI, H.; KIM, H.-k.; KANG, H.-j.; KIM, K.; KIUMI, A.; KANG, B.-H. (Ed.). *Software Engineering, Business Continuity, and Education*. Springer Berlin Heidelberg, 2011, (Communications in Computer and Information Science, v. 257). p. 147–158. ISBN 978-3-642-27206-6. Disponível em: <http://dx.doi.org/10.1007/978-3-642-27207-3_15>.

BÖLLERT, K. On weaving aspects. In: *Proceedings of the Workshop on Object-Oriented Technology*. London, UK, UK: Springer-Verlag, 1999. p. 301–302. ISBN 3-540-66954-X. Disponível em: <http://dl.acm.org/citation.cfm?id=646779.705621>.

BRUNTINK, M.; DEURSEN, A. van; ENGELEN, R. van; TOURWE, T. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, IEEE Press,

Piscataway, NJ, USA, v. 31, n. 10, p. 804–818, out. 2005. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2005.114>.

CARNEIRO, G. d. F.; SILVA, M.; MARA, L.; FIGUEIREDO, E.; SANT'ANNA, C.; GARCIA, A.; MENDONCA, M. Identifying code smells with multiple concern views. In: *Proceedings of the 2010 Brazilian Symposium on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2010. (SBES '10), p. 128–137. ISBN 978-0-7695-4273-7. Disponível em: <http://dx.doi.org/10.1109/SBES.2010.21>.

CECCATO, M.; MARIN, M.; MENS, K.; MOONEN, L.; TONELLA, P.; TOURWÉ, T. Applying and combining three different aspect mining techniques. *Software Quality Journal*, Kluwer Academic Publishers, v. 14, n. 3, p. 209–231, 2006. ISSN 0963-9314. Disponível em: <http://dx.doi.org/10.1007/s11219-006-9217-3>.

CHAVEZ, C.; KULESZA, U.; SOARES, S.; BORBA, P.; LUCENA, C.; MASIERO, P.; SANT'ANNA, C.; PIVETA, E.; FERRARI, F.; CASTOR, F.; COELHO, R.; SILVA, L.; ALVES, V.; MENDONCA, N.; FIGUEIREDO, E.; CAMARGO, V.; SILVA, C.; PIRES, P.; BATISTA, T.; CACHO, N.; STAA, A. von; LEITE, J.; SILVEIRA, F.; LEMOS, O.; PENTEADO, R.; DELICATO, F.; BRAGA, R.; VALENTE, M.; RAMOS, R.; BONIFACIO, R.; ALENCAR, F.; CASTRO, J. The aosd research community in brazil and its crosscutting impact. In: *Software Engineering (SBES), 2011 25th Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 72–81.

COADY, Y.; KICZALES, G.; FEELEY, M.; SMOLYN, G. Using aspectc to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 26, n. 5, p. 88–98, set. 2001. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/503271.503223>.

CZIBULA, G.; COJOCAR, G. S.; CZIBULA, I. G. Evaluation Measures for Partitioning based Aspect Mining Techniques. *International Journal of Computers, Communications & Control*, VI, n. 1, p. 72–80, March 2011.

DAGENAIS, B.; OSSHER, H. Mismar: A new approach to developer documentation. In: *Companion to the proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE COMPANION '07), p. 47–48. ISBN 0-7695-2892-9. Disponível em: <http://dx.doi.org/10.1109/ICSECOMPANION.2007-.51>.

DELFIM, F. M. *Uma Abordagem Usando Visualização de Software como Apoio à Refatoração para Aspectos*. Dissertação (Dissertação de Mestrado) — Universidade Estadual Paulista Júlio de Mesquita Filho (UNESP), 2013.

DELFIM, F. M.; GARCIA, R. E. Multiple coordinated views to support aspect mining using program slicing. In: *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE '13)*. [S.l.: s.n.], 2013. p. 531–536.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. Object-oriented reengineering: patterns and techniques. In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [S.l.: s.n.], 2005. p. 723–724. ISSN 1063-6773.

DIEHL, S. (Ed.). *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, v. 2269 de *Lecture Notes in Computer Science*, (Lecture Notes in Computer Science, v. 2269). [S.l.]: Springer, 2002. ISBN 3-540-43323-6.

DUCASSE, S.; RIEGER, M.; DEMEYER, S. A language independent approach for detecting duplicated code. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. [S.l.: s.n.], 1999. p. 109–118. ISSN 1063-6773.

DURELLI, V. H. S. *Reengenharia Iterativa do Framework GREN*. Dissertação (Dissertação de Mestrado) — PPGCC - Universidade Federal de São Carlos (UFSCAR), 2008.

Eclipse Foundation. *Eclipse - The Eclipse Foundation open source community website*. 2013. http://eclipse.org/. Acesso em: outubro de 2013.

ESTATCAMP. *Boxplot*. 2013. http://www.portalaction.com.br/content/31-boxplot. Acesso em: outubro de 2013.

ESTATCAMP. *Cálculo e interpretação do p-valor*. 2013. http://www.portalaction.com.br/content/64-teste-de-shapiro-wilk. Acesso em: outubro de 2013.

ESTATCAMP. *Cálculo e interpretação do p-valor*. 2013. http://www.portalaction.com.br/539-512-cálculo-e-interpretação-do-p-valor. Acesso em: outubro de 2013.

ESTATCAMP. *Diagnóstico de Outiliers*. 2013. http://www.portalaction.com.br/923-diagnóstico-de-outliers. Acesso em: outubro de 2013.

ESTATCAMP. *Portal Action*. 2013. http://www.portalaction.com.br/. Acesso em: outubro de 2013.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-48567-2.

HANNEMANN, J.; KICZALES, G. Overcoming the prevalent decomposition of legacy code. In: *In Workshop on Advanced Separation of Concerns*. [S.l.: s.n.], 2001.

JHOTDRAW.ORG. *JHotDraw*. 2013. http://www.jhotdraw.org/. Acesso em: outubro de 2013.

KAMIYA, T.; KUSUMOTO, S.; INOUE, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 28, n. 7, p. 654–670, jul. 2002. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2002.1019480>.

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

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of aspectj. In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK, UK: Springer-Verlag, 2001. (ECOOP '01), p. 327–353. ISBN 3-540-42206-4. Disponível em: <http://dl.acm.org/citation.cfm?id=646158.680006>.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: AKşIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming*. Springer Berlin Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 220–242. ISBN 978-3-540-63089-0. Disponível em: <http://dx.doi.org/10.1007/BFb0053381>.

KRINKE, J. Identifying similar code with program dependence graphs. In: *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, 2001. (WCRE '01), p. 301–. ISBN 0-7695-1303-4. Disponível em: <http://dl.acm.org/citation.cfm?id=832308.837142>.

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

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

MALETIC, J.; MOSORA, D.; NEWMAN, C.; COLLARD, M.; SUTTON, A.; ROBINSON, B. Mosaicode: Visualizing large scale software: A tool demonstration. In: *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*. [S.l.: s.n.], 2011. p. 1–4.

MARCUS, A.; FENG, L.; MALETIC, J. I. 3d representations for software visualization. In: *Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2003. (SoftVis '03), p. 27–ff. ISBN 1-58113-642-0. Disponível em: <http://doi.acm.org/10.1145/774833.774837>.

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

MAYRAND, J.; LEBLANC, C.; MERLO, E. Experiment on the automatic detection of function clones in a software system using metrics. In: *Software Maintenance 1996, Proceedings., International Conference on*. [S.l.: s.n.], 1996. p. 244–253. ISSN 1063-6773.

MCFADDEN, R.; MITROPOULOS, F. Aspect mining using model-based clustering. In: *Southeastcon, 2012 Proceedings of IEEE*. [S.l.: s.n.], 2012. p. 1–8. ISSN 1091-0050.

MICROSOFT, O. *Microsoft Excel*. 2013. http://office.microsoft.com/pt-br/excel/. Acesso em: outubro de 2013.

NGUYEN, T. T.; NGUYEN, H. V.; NGUYEN, H. A.; NGUYEN, T. N. Aspect recommendation for evolving software. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 361–370. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985843>.

OLIVEIRA, M. de; LEVKOWITZ, H. From visual data exploration to visual data mining: a survey. *Visualization and Computer Graphics, IEEE Transactions on*, v. 9, n. 3, p. 378–394, 2003. ISSN 1077-2626.

PARREIRA JÚNIOR, P. A. *Recuperação de Modelos de Classes Orientados a Aspectos a partir de Sistemas Orientados a Objetos Usando Refatorações de Modelos*. Dissertação (Dissertação de Mestrado) — PPGCC - Universidade Federal de São Carlos (UFSCAR), 2011.

PARREIRA JÚNIOR, P. A.; CAMARGO, V. V. d.; PENTEADO, R. A. D.; COSTA, H. A. X. Uma abordagem iterativa para identificação de interesses transversais com o apoio de modelos de classes anotados. In: *LATIN AMERICAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT - LA-WASP, Salvador, Brasil, 2010*. [S.l.: s.n.], 2010a.

PARREIRA JÚNIOR, P. A.; COSTA, H. A. X.; CAMARGO, V. V. d.; PENTEADO, R. A. D. Experiment on the automatic detection of function clones in a software system using metrics. In: *WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA - WMSWM, Belém, Brasil, 2010*. [S.l.: s.n.], 2010b.

PIEFEL, M. A common metamodel for code generation. In: *Proceedings of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications. I I I S*. [S.l.: s.n.], 2006.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 5th. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681.

RIEL, A. J. *Object-Oriented Design Heuristics*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 020163385X.

ROBILLARD, M. *The ConcernMapper Eclipse Plug-in*. 2013. http://www.cs.mcgill.ca/ martin/cm/. Acesso em: outubro de 2013.

ROBILLARD, M.; MURPHY, G. Concern graphs: finding and describing concerns using structural program dependencies. In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. [S.l.: s.n.], 2002. p. 406–416.

ROBILLARD, M. P.; MURPHY, G. C. Feat: a tool for locating, describing, and analyzing concerns in source code. In: *Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003. (ICSE '03), p. 822–823. ISBN 0-7695-1877-X. Disponível em: <http://dl.acm.org/citation.cfm?id=776816.776969>.

ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 16, n. 1, fev. 2007. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/1189748.1189751>.

ROBILLARD, M. P.; WEIGAND-WARR, F. Concernmapper: simple view-based separation of scattered concerns. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. New York, NY, USA: ACM, 2005. (eclipse '05), p. 65–69. ISBN 1-59593-342-5. Disponível em: <http://doi.acm.org/10.1145/1117696.1117710>.

SCHOTS, M.; WERNER, C.; MENDONCA, M. Awareness and comprehension in software/systems engineering practice and education: Trends and research directions. In: *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. [S.l.: s.n.], 2012. p. 186–190.

SHNEIDERMAN, B. Tree visualization with tree-maps: A 2-d space-filling approach. *ACM Transactions on Graphics*, v. 11, p. 92–99, 1991.

SILVA A. N.; CARNEIRO, G. F. Propondo uma arquitetura para ambientes interativos baseados em multiplas visões. *II Workshop Brasileiro de Visualização de Software (WBVS), 2012, Natal - RN. Congresso Brasileiro de Software (CBSOFT)*, 2012.

SPINCZYK, O.; GAL, A.; SCHRöDER-PREIKSCHAT, W. Aspectc++: an aspect-oriented extension to the c++ programming language. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002. (CRPIT '02), p. 53–60. ISBN 0-909925-88-7. Disponível em: <http://dl.acm.org/citation.cfm?id=564092-.564100>.

TANNER, O. J. F.; PENTEADO, R. A. D.; VIANA, M. C.; CARNEIRO, G. F. Metamodelo para integração de ferramentas de mineraćão de interesses transversais e de visualização de software. In: *WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA - WMSWM, Salvador, Brasil, 2013*. [S.l.: s.n.], 2013.

TREUDE, C.; STOREY, M.-A. The implications of how we tag software artifacts: exploring different schemata and metadata for tags. In: *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*. New York, NY, USA: ACM, 2010. (Web2SE '10), p. 12–13. ISBN 978-1-60558-975-6. Disponível em: <http://doi.acm.org/10.1145/1809198.1809203>.

VIANA, M.; PENTEADO, R.; PRADO, A. do; DURELLI, R. F3T: From Features to Frameworks Tool. In: *27th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2013.

WARE, C. *Information Visualization: Perception for Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608192.

WETTEL, R.; LANZA, M. Visualizing software systems as cities. In: *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*. [S.l.: s.n.], 2007. p. 92–99.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

XSTREAM. *XStream*. 2013. http://xstream.codehaus.org/index.html. Acesso em: outubro de 2013.

ZHANG, C.; JACOBSEN, H.-A. Mining crosscutting concerns through random walks. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 38, n. 5, p. 1123–1137, 2012. ISSN 0098-5589.

GLOSSÁRIO

AI – *Ambiente Integrado*

ANI – *Ambiente Não Integrado*

CC – *Crosscutting Concerns*

CM – *ConcernMapper*

CSV – *Comma-Separated Values*

CV – *Conjunto de Visões*

IMM – *Indications Metamodel*

IT – *Interesse Transversal*

MIT – *Mineração de Interesses Transversais*

MT – *Mining Tool*

OA – *Orientação a Aspectos*

OO – *Orientação a Objetos*

POA – *Programação Orientação a Aspectos*

SQL – *Structured Query Language*

XML – *eXtensible Markup Language*

Apendice A

FORMULÁRIO DE EXECUÇÃO DO EXPERIMENTO DE INTEGRAÇÃO

1. Dados do Participante

Nome:	
Formato:	( ) IMM ( ) CM

2. Execução do Experimento

Preenchido pelo Participante		**Preenchido pelo Proponente do Experimento**
Hora Início: ___ : ___	Hora Término: ___ : ___	Tempo I (minutos):

Interrupções

Preenchido pelo Participante		**Preenchido pelo Proponente do Experimento**
Hora Início: ___ : ___	Hora Término: ___ : ___	T1 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T2 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T3 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T4 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T5 (minutos):
TOTAL		Tempo II (minutos):

Tempo total do participante (Tempos: I - II):	

3. Observações

Apendice B

DESCRITIVO DO FORMATO CM

- Todo arquivo CM deve ser escrito sem nenhum tipo de quebra de linha e sem nenhum tipo de espaço entre as estruturas;

- Todo arquivo CM deve começar com:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

- Todos os interesses devem estar dentro de uma *tag* `<model></model>`

- Um interesse é representado pela *tag* `<concern></concern>` de acordo com o exemplo:

```
<concern name="X"></concern>
```

- Todo indício é representado pela *tag* `<element/>` de acordo com:
  - `<element degree="100" id="" type=""/>`
  - **degree** é igual a 100
  - **id** representa o método ou o atributo afetado pelo interesse transversal
  - **type** indica se é um método (`method`) ou um atributo (`field`)

- O campo id deve conter:
  - Em caso de método: "=NomeProjeto/src<NomePacote{NomeClasse.java[NomeClasse~NomeMetodoIdParametro"
  - Em caso de atributo: "=NomeProjeto/src<NomePacote{NomeClasse.java[NomeClasse^NomeAtributo"

]

- IdParametro deve seguir as regras da tabela abaixo:

Java	**CM**
boolean	~Z
char	~C
byte	~B
short	~S
int	~I
long	~J
float	~F
double	~D
NomeClasse	~QNomeClasse;
boolean[]	~\[Z
char[]	~\[C
byte[]	~\[B
short[]	~\[S
int[]	~\[I
long[]	~\[J
float[]	~\[F
double[]	~\[D
NomeClasse[]	~\[QnomeClasse;

Apendice C

FORMULÁRIO DE CONSENTIMENTO

Título do projeto e Propósito

Experimento comparativo entre uma abordagem não integrada e uma abordagem integrada de Mineração de Interesses Transversais e Visualização de Software para a identificação de indícios de interesses transversais em um software.

O objetivo do experimento é comparar o tempo e a quantidade de indícios de interesses transversais encontrados em um software utilizando uma abordagem não integrada e uma abordagem integrada de Mineração de Interesses Transversais e Visualização de Software. A partir da comparação serão verificadas as vantagens e desvantagens de se utilizar a abordagem integrada.

Declaração de idade
Eu declaro que sou maior de 18 anos de idade e que desejo participar do experimento, a ser realizado no laboratório do grupo de pesquisa AdvanSE e que será conduzido por Oscar José Fernandes Tanner, estudante de mestrado do PPGCC-UFSCar.

Procedimentos
O experimento envolve três etapas. Na primeira etapa será conduzido um treinamento que aborda os conceitos de Orientação a Aspectos, Mineração de Interesses Transversais e Visualização de Software, bem como ferramentas que implementem essas técnicas. Na segunda etapa será realizado um experimento piloto com dois cenários e dois sistemas exemplo: a) identificação dos indícios de interesses transversais por meio de uma abordagem não integrada de Mineração de Interesses Transversais e Visualização de Software e, b) identificação dos indícios de interesses transversais por meio de uma abordagem integrada de Mineração de Interesses Transversais e Visualização de Software. Na terceira etapa serão utilizados os dois cenários anteriores, com outros dois diferentes sistemas, com mesmo grau de similaridade dos apresentados na segunda etapa.

Confidência
Todas as informações coletadas no experimento são confidenciais, e meu nome não será identificado em tempo algum.

Benefícios, liberdade para retirar-se.
Estou consciente que não terei remuneração pela participação no experimento, mas que o estudo ajudará a aprender mais sobre o uso de Mineração de Interesses Transversais integrada à Visualização de Software. Eu entendo que sou livre para perguntar ou para retirar minha participação em qualquer tempo sem penalidade, e que eu terei acesso aos resultados principais do experimento.

Responsáveis

Oscar José Fernandes Tanner
Univ. Federal de São Carlos
São Carlos-SP

Profª Drª Rosângela Ap. D. Penteado
Univ. Federal de São Carlos
São Carlos-SP

_____
Nome do participante

_____
Assinatura do participante Data

Apendice D

QUESTIONÁRIO DE PERFIL DO PARTICIPANTE

QUESTIONÁRIO INDIVIDUAL PARA OS PARTICIPANTES DO EXPERIMENTO DA DISCIPLINA

Nome:
Data:

Conhecimento dos conceitos de Orientação a Objetos:
( ) Alto
( ) Médio
( ) Baixo

Nível de conhecimento em relação à linguagem Java:
( ) Básico – aproximadamente 1 ano.
( ) Intermediário – mais de 2 anos.
( ) Avançado – 5 anos ou mais.

Conhecimento dos conceitos de Orientação a Aspectos (interesses, pointcuts e advices):
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento dos conceitos de Mineração de Interesses Transversais
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento dos conceitos de Visualização de Software
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento da ferramenta ComSCId:
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento da ferramenta ConcernMapper:
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento da ferramenta FINT:
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento da ferramenta XSCan:
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Conhecimento da ferramenta SourceMiner:
( ) Alto
( ) Médio
( ) Baixo (somente o que foi apresentado nesta disciplina)

Nível de conhecimento em relação à linguagem AspectJ:
( ) Baixo (somente o que foi apresentado nesta disciplina)
( ) Intermediário – mais de 1 ano.
( ) Avançado – 2 anos ou mais.

Em qual das categorias abaixo você se encaixa como desenvolvedor?
( ) sistemas triviais desenvolvidos durante o curso de graduação.
( ) sistemas de médio porte; com aproximadamente 7-20 mil linhas de código.
( ) sistemas de grande porte; com aproximadamente 20-50 mil linhas de código.

Classifique seu nível de habilidade com o ambiente Eclipse:
( ) Não tenho experiência alguma
( ) Tenho alguma experiência
( ) Tenho bastante experiência