UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VICTOR SOBREIRA

UMA ABORDAGEM PARA COMPREENSÃO DE PROGRAMAS BASEADA NA LOCALIZAÇÃO DE CARACTERÍSTICAS EM CÓDIGO FONTE

UBERLÂNDIA - MG
2008

VICTOR SOBREIRA

UMA ABORDAGEM PARA COMPREENSÃO DE PROGRAMAS BASEADA NA LOCALIZAÇÃO DE CARACTERÍSTICAS EM CÓDIGO FONTE

Dissertação de Mestrado apresentada à Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Orientador:
Prof. Marcelo de Almeida Maia

UBERLÂNDIA - MG
SETEMBRO/2008

Dados Internacionais de Catalogação na Publicação (CIP)

S677a Sobreira, Victor, 1981-
Uma abordagem para compreensão de programas baseada na localização de características em código fonte / Victor Sobreira. – 2008.
197 f. : il.

Orientador: Marcelo de Almeida Maia.
Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.

Inclui bibliografia.

1. 1. Software - Manutenção - Teses. I. Maia, Marcelo de Almeida. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3.06

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO

FICHA DE APROVAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada **"Uma abordagem para compreensão de programas baseada na localização de características em código fonte"** por **Victor Sobreira** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 12 de Setembro de 2008.

Orientador:

Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia UFU/MG

Banca Examinadora:

Prof. Dr. Stéphane Julia
Universidade Federal de Uberlândia UFU/MG

Prof. Dr. Roberto da Silva Bigonha
Universidade Federal de Minas Gerais UFMG/MG

À minha querida bisavó Geraldina

(*in memorian*).

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado a possibilidade e todas as condições para a realização deste trabalho.

Agradeço ao meu orientador, Marcelo, por ter acreditado e apoiado a realização da pesquisa sempre de forma paciente e solícita. Certamente, sem sua dedicação e atenção tudo isto não teria sido possível.

Agradeço a meus pais, Izilda e Tadeu, por sempre apoiarem meus estudos e terem me estimulado desde o início a acreditar em minha capacidade de realização e superação.

Agradeço à minha mais que namorada, Mirella, que viveu comigo tudo isso e sempre esteve do meu lado em todas as situações, me apoiando, me ajudando e me estimulando.

Agradeço à coordenação pela oportunidade, à CAPES pelo financiamento parcial e aos professores da banca pelas sugestões e correções.

Por fim, agradeço a todos que contribuíram direta ou indiretamente para a realização e conclusão deste trabalho.

"Faz apenas o que amas e será feliz. Aquele que faz o que ama está benditamente condenado ao sucesso, o qual virá na hora certa, pois o que deve ser será e chegará naturalmente."

(CABRAL, 2008, tradução nossa)

RESUMO

SOBREIRA, Victor. **Uma abordagem para compreensão de programas baseada na localização de características em código fonte;** Orientador: Prof. Marcelo de Almeida Maia; Uberlândia-MG; UFU, 2008, 197 fl., Dissertação de Mestrado.

Características são conceitos importantes para o entendimento de requisitos de *software*. Entretanto, descobrir onde uma característica de interesse está localizada no código fonte é uma tarefa desafiadora porque, em geral, o código é modularizado de maneira não orientada a características e assim as características tendem a estar espalhadas pelo código fonte. Este trabalho propõe um método e uma ferramenta chamada *Featincode* para a análise do espalhamento de características através da interpretação gráfica da interseção entre características e elementos do código fonte. A ferramenta coleta e representa rastros de programas *multi-threaded* para as características selecionadas pelo desenvolvedor e mostra algumas matrizes que ajudam a analisar onde as características estão implementadas. O método e a ferramenta proposta são validadas com a análise de algumas características da ferramenta *CASE ArgoUML*. A conclusão é que a abordagem pode reduzir o esforço para compreender onde as características estão implementadas e quais elementos do código são específicos de uma característica. O método e a ferramenta podem ser aplicados para auxiliar em atividades de compreensão, manutenção e evolução de *software*.

Palavras-chave: Manutenção de Software, Compreensão de Programas, Visualização de Software, Localização de Características, Análise Dinâmica.

ABSTRACT

SOBREIRA, Victor. **An approach to software comprehension based on feature location in source code;** Master's Thesis Supervisor: Prof. Marcelo de Almeida Maia; Uberlândia-MG; UFU, 2008, 197 fl., Master Thesis.

Features are an important concept for understanding software requirements. However, discovering where a desired feature is located in the source code is a challenging task because the code is generally not modularized in a feature-fashioned way, and so, features tend to be scattered over the source code. This work proposes a method and a software tool called *Featincode* for analyzing feature scattering through the graphical interpretation of the intersection between feature elements and source code elements. The tool collects and represents trace events of multi threaded programs for developer selected features and show some matrices that help analyzing where those features are implemented. The proposed method and tool are validated with the analysis of some features of the *ArgoUML* CASE tool. The conclusion is that our approach can reduce the effort for comprehending where features are implemented and which source code is specific to a feature. The method and tool can be applied to help software comprehension, maintenance and evolution activities.

Keywords: Software Maintenance, Program Comprehension, Software Visualization, Feature Location, Dynamic Analysis.

LISTA DE FIGURAS

Figura 2.1: Modelo de Características para um Carro, adaptado de (CZARNECKI; EISENECKER, 2000).....9
Figura 2.2: Notações para modelagem de características.....13
Figura 2.3: Exemplo da execução de redirecionamento de chamadas num sistema de chaveamento telefônico extraído de (WILDE, 1994).....17
Figura 2.4: a) Exemplo da técnica Dotplot sobre a pequena citação "to be or not to be"; b) Aplicação da técnica sobre um coletânea das obras de Shakespeare.....23
Figura 2.5: Padrão Square.....25
Figura 2.6: Padrão Diagonals.....25
Figura 2.7: Padrão Light Cross.....25
Figura 2.8: Padrão Broken Diagonals.....25
Figura 2.9: Padrão Reordered Squares.....25
Figura 2.10: Padrão Reordered Diagonals.....25
Figura 2.11: Duas versões de xmh, um sistema com 20.000 linhas de código C contidas em 53 arquivos.....26
Figura 2.12: 290.000 nomes de arquivos e o padrão Light Cross indicando uma seqüência de 15.000 arquivos com nomes únicos.....27
Figura 2.13: Duas versões do X Toolkit, envolvendo 66.600 linhas de código C.....27
Figura 2.14: Inicializações reordenadas num arquivo com 2500 linhas de código definindo um objeto gráfico...28
Figura 2.15: Duas versões do X Window System composto por 1.9 milhão de linhas de código.....29
Figura 3.1: Atividades para a utilização da ferramenta Featincode.....31
Figura 3.2: Interface de controle do extrator de rastros executando em paralelo com a aplicação Eye of The Tjger. 32
Figura 3.3: Interface principal da ferramenta de análise.....33
Figura 3.4: Visualizador do modelo de código fonte e controlador dos elementos exibidos na matriz.....36
Figura 3.5: Visualizador do modelo de características e passos de execução e controlador dos elementos da matriz.....41
Figura 3.6: Exemplo de modelo de características gerado automaticamente pela ferramenta de extração.....42
Figura 3.7: Exemplo de modelo de características editado pelo usuário.....43
Figura 3.8: Visualizador do rastro de execução mostrando o espalhamento de elementos do modelo de código fonte e do modelo de características.....44
Figura 3.9: Visualizador de métricas.....46
Figura 3.10: Matrizes de visualização de métricas. a) Matriz Cobertura de Características; b) Matriz Cobertura de Elementos do Código Fonte; c) Matriz Similaridade entre Características; d) Matriz Similaridade entre Elementos do Código Fonte.....51
Figura 3.11: Mapas de cores para as escalas Hot Cold, Gray e Discrete Levels.....51
Figura 3.12: Detalhes para a célula selecionada correspondente ao pacote sample.game e a característica "Game Play".....54
Figura 3.13: Janela de configuração dos filtros para as linhas da matriz.....55
Figura 3.14: Visão Geral da Arquitetura do Featincode.....56
Figura 3.15: Parte do código responsável pela captura dos eventos no sistema alvo sendo executado.....57
Figura 3.16: Exemplo de trecho de arquivo "data.trace" com rastro de execução gerado por uma thread.....58
Figura 3.17: Exemplo de trecho de arquivo com passos para um roteiro de execução definido previamente.....59
Figura 3.18: Exemplo de trecho de arquivo contendo as marcações dos passos de execução.....59
Figura 3.19: Eventos redundantes removidos do rastro com base no algoritmo implementado e adaptado de (HAMOU-LHADJ; LETHBRIDGE, 2002).....60
Figura 3.20: Classes de indexação no pacote traceanalyser.processing.indexers.....64
Figura 3.21: Subpacotes e classes do pacote traceanalyser.processing.metrics.....64
Figura 3.22: Padrão Proxy (GAMMA et al, 1995) utilizado como base para a estratégia de Cache.....65
Figura 3.23: Estrutura de classes do pacote traceanalyser.model.source_code.....66
Figura 3.24: Estrutura de classes do pacote traceanalyser.model.features.....67
Figura 3.25: Estrutura de classes do pacote traceanalyser.model.trace.....67
Figura 4.1: Parte do Modelo de Características para a aplicação Eye of the Tjger.....77
Figura 4.2: Relacionamento entre características e passos de execução para o Roteiro de Execução 1.....78
Figura 4.3: Modelo de código fonte extraído do rastro de execução do Roteiro 1 apresentando estatísticas de tempo de execução para o pacote tjger.....82
Figura 4.4: Passos de execução referentes ao Roteiro 1.....83
Figura 4.5: Passos de execução referentes ao Roteiro 1, agrupados segundo o modelo de características.....83

Figura 4.6: Espalhamento da inicialização e ajuste da aparência da aplicação para o estilo Windows referente à execução do Roteiro 1 (Opções de Configuração) da aplicação Eye of the Tjger.....84
Figura 4.7: Visualização dos eventos no rastro associados aos pacotes principais (hgb, sample, tjger) (verde) e as características de mudança de língua (azul) da aplicação Eye of the Tjger na execução do Roteiro 1.....87
Figura 4.8: Matrizes com métricas de cobertura de características mostrando a representatividade dos pacotes sobre as características no rastro da Thread-2 do Roteiro 1 em termos de eventos (NSEPF), métodos (NPMF), classes (NPCF) e pacotes (NPPF), respectivamente.....88
Figura 4.9: Matrizes com métrica de cobertura de característica NSEPF nas diferentes escalas de cores mostrando a representatividade dos pacotes sobre as características em termos de eventos no rastro da Thread-2 do Roteiro 1.....89
Figura 4.10: Matriz com métrica de cobertura de característica NSEPF com linhas ordenadas mostrando a representatividade dos pacotes sobre as características em termos de eventos no rastro da Thread-2 do Roteiro 1. 90
Figura 4.11: Similaridade entre os pacotes em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da Thread-2 e na métrica NSFP.....92
Figura 4.12: Similaridade entre classes do pacote sample em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da Thread-2 e na métrica NSFP.....93
Figura 4.13: Similaridade entre classes do pacote tjger em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da Thread-2 e na métrica NSFP.....95
Figura 4.14: Similaridade entre classes do pacote hgb em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da Thread-2 e na métrica NSFP.....96
Figura 4.15: Similaridade entre classes dos pacote hgb, sample e tjger em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da Thread-2 e na métrica NSFP.....98
Figura 4.16: Similaridade entre características quanto ao número de pacotes (NSPF), classe (NSCF) e métodos (NSMF) compartilhados durante a execução do Roteiro 2 e para os rastros da Thread-2.....99
Figura 4.17: Similaridade entre os passos de execução quanto ao número de pacotes (NSPF), classe (NSCF) e métodos (NSMF) compartilhados durante a execução do Roteiro 2 e para os rastros da Thread-2.....100
Figura 4.18: Similaridade entre passos de execução (NSMF) com destaque para a replicação das jogadas na partida para o Roteiro 2 formando o padrão Dotplot Diagonals.....101
Figura 5.1: Modelo de Características com principais características do ArgoUML.....105
Figura 5.2: Modelo de Características do ArgoUML com algumas expansões nos nós principais.....105
Figura 5.3: Janela principal do ArgoUML-0.19.3.....106
Figura 5.4: Eventos associados à Execução 1 da Inicialização do ArgoUML.....112
Figura 5.5: Rastro gerado para a Execução 1 de “Auto-Crítica 1” destacando a inicialização e a espera dos 10 segundos após desabilitar as auto-críticas do ArgoUML.....113
Figura 5.6: Rastro gerado para a Execução 1 de “Auto-Crítica 2” destacando a inicialização e a espera dos 15 segundos com as auto-críticas do ArgoUML habilitadas.....113
Figura 5.7: Participação característica “Create Class Diagram by menu” no rastro para Execução 1 da Sequência 1.....114
Figura 5.8: Participação da característica “Create Class Diagram by shortcut” no rastro para Execução 1 da Sequência 1 (Thread-2).....114
Figura 5.9: Participação da característica “Create Class Diagram by toolbar” no rastro para Execução 1 da Sequência 1 (Thread-2).....114
Figura 5.10: Participação dos pacotes org e ru no rastro da Execução 1 da Seqüência 1.....117
Figura 5.11: Matrizes de visualização mostrando o número distinto de elementos do código (pacotes, classes e métodos) e eventos derivados do rastro gerado pela Thead-2 na Execução 1 da Sequência 1, enfatizando os pacotes principais envolvidos no rastro.....120
Figura 5.12: Matrizes de visualização mostrando o número distinto de elementos do código (pacotes, classes e métodos) e eventos derivados do rastro gerado pela Thead-2 na Execução 1 da Seqüência 1, enfatizando apenas pacotes que contém elementos participantes nas características.....123
Figura 5.13: Matrizes de visualização mostrando o número distinto de métodos e eventos derivados do rastro gerado pela Thead-2 na Execução 1 da Seqüência 1, enfatizando apenas classes participantes nas características (Escala Hot Cold Log).....127
Figura 5.14: Classes participantes nas características ordenadas segundo a coloração das células e na escala Hot Cold Log.....128
Figura 5.15: Métodos das classes UMLClassDiagram e Project participantes nas características.....129
Figura 5.16: Redução no escopo da análise (pacotes) para Execução 1 da Seqüência 1.....133
Figura 5.17: Redução no escopo da análise (classes e interfaces) para a Execução 1 da Seqüência 1.....133
Figura 5.18: Redução no escopo da análise (métodos) para a Execução 1 da Seqüência 1.....133
Figura 5.19: Participação das características no rastro da Execução 1 do Roteiro 2.....140

Figura 5.20: Participação em número de eventos e métodos nas características (Execução 1, Roteiro 1).....143
Figura 5.21: Similaridade entre características observada em expansões sucessivas das matrizes Característica x Característica.....147
Figura 5.22: Similaridade entre características destacando na diagonal principal as células relacionadas à criação de cada diagrama.....149
Figura 5.23: Redução no escopo para a análise de Pacotes, Classes e Métodos.....150
Figura 5.24: Influência da ordem de execução sobre a interpretação das matrizes (métrica NSEPF e escala Hot Cold Log).....152
Figura 7.1: Aplicação da técnica Dotplot a uma seqüência de eventos do rastro de execução (apenas ilustrativa, não correspondendo a dados reais).....171

LISTA DE TABELAS

Tabela 3.1: Métricas de Cobertura e Similaridade para Elementos do Código Fonte e Características.....47
Tabela 3.2: Compressão obtida sobre um rastro de execução gerado pelo ArgoUML.....61
Tabela 4.1: Informações x Fonte na ferramenta Featincode.....73
Tabela 4.2: Roteiro de Execução 1 envolvendo algumas características de configuração do Eye of the Tjger.....78
Tabela 4.3: Roteiro de Execução 2 com a seqüência de uma partida de Eye of The Tjger.....79
Tabela 4.4: Cobertura do código pelos roteiros de execução.....80
Tabela 4.5: Estatísticas de tempo de execução para o Roteiro 2 (Seqüência da Partida).....81
Tabela 4.6: Estatísticas de tempo de execução para o Roteiro 1 (Opções de Configuração).....81
Tabela 4.7: Relação entre pacotes e quantidade de passos de execução compartilhados no rastro da Thread-2 para o Roteiro 2.....93
Tabela 5.1: Métricas extraídas do código fonte do ArgoUML 0.19.3 com o plugin do Eclipse PMD.....104
Tabela 5.2. Pontos similares e distintos entre os formas de interação para criação de diagramas de classe no ArgoUML.....108
Tabela 5.3: Número de eventos gerados no rastro para as execuções relacionadas ao primeiro contexto: Características Muito Similares.....111
Tabela 5.4. Classes presentes no rastro gerado pela Thread-3 na Execução da 1 Sequência 4.....115
Tabela 5.5.Classes presentes no rastro gerado pela Thread-3 na Execução 2 da Sequência 3 (destacando classes e pacotes adicionais em relação à Tabela 5.5).....116
Tabela 5.6: Participação dos pacotes nas threads (Execução 1, Seqüência 1).....118
Tabela 5.7: Pacotes compartilhados - Thread-2, Seq-1, Exec-1.....121
Tabela 5.8: Pacotes compartilhados (%) - Thread-2, Seq-1, Exec-1.....121
Tabela 5.9: Classes compartilhadas - Thread-2, Seq-1, Exec-1.....121
Tabela 5.10: Classes compartilhadas (%) - Thread-2, Seq-1, Exec-1.....121
Tabela 5.11: Métodos compartilhados - Thread-2, Seq-1, Exec-1.....121
Tabela 5.12: Métodos compartilhados (%) - Thread-2, Seq-1, Exec-1.....121
Tabela 5.13: Eventos compartilhados - Thread-2, Seq-1, Exec-1.....121
Tabela 5.14: Eventos compartilhados (%) - Thread-2, Seq-1, Exec-1.....121
Tabela 5.15: Classes exclusivas das características exercitadas na Execução 1 da Seqüência 1 (Thread-2).....125
Tabela 5.16: Métodos exclusivos das características.....129
Tabela 5.17: Métricas extraídas do rastro (Execução 1, Seqüência 1).....131
Tabela 5.18: Cobertura dos pacotes no código fonte pelos elementos presentes no rastro (Execução 1, Seqüência 1).....131
Tabela 5.19: Projeção geral dos elementos do rastro sobre o total de elementos do código (Execução 1, Sequência 1).....131
Tabela 5.20: Cobertura dos pacotes no código fonte pelos elementos presentes no rastro gerados pela Thread-2 na Execução 1 da Seqüência 1 e associados às características exercitadas.....132
Tabela 5.21: Métricas extraídas do rastro gerado pela Thread-2 na Execução 1 da Seqüência 1.....132
Tabela 5.22: Projeção dos elementos do rastro gerados pela Thread-2 na Execução 1 da Seqüência 1 e associados às características exercitadas sobre o total de elementos do código.....132
Tabela 5.23: Elementos comuns entre diagramas UML.....136
Tabela 5.24: Número de eventos gerados pelos roteiros de execução do Contexto 2.....140
Tabela 5.25: Pacotes com maior concentração em cada característica.....143
Tabela 5.26: Classes exclusivas das características.....144
Tabela 5.27: Classes com métodos exclusivos.....145
Tabela 5.28: Classes com métodos exclusivos.....146
Tabela 5.29: Classes relacionadas à execução de dois passos de execução comuns entre o desenho de diagramas de classe.....150

Sumário

Capítulo 1. Introdução....1
1 Contextualização....1
2 Exposição do Problema....2
3 Hipóteses e Proposta....3
4 Objetivos....3
5 Justificativas....4
6 Visão Geral da Dissertação....4
Capítulo 2. Referencial Teórico....6
1 Introdução....6
2 Análise de Características....6
2.1 Origens e Motivações....6
2.2 Conceito de Característica....7
2.3 Modelo de Características....8
3 Modelagem de Características....10
3.1 Fontes de informação....10
3.2 Estratégias de identificação....11
3.3 Processo de Modelagem....12
3.4 Criação do Modelo de Características....12
4 Localização de Características....14
4.1 Motivações....15
4.2 Problemas....16
4.3 Soluções....16
5 Análise Dinâmica....19
5.1 Planejamento da execução....19
5.1.1 Definição dos roteiros de execução....20
5.1.2 Extração de rastro único x Extração de vários rastros por característica....20
5.2 Mecanismos de Coleta....20
5.2.1 Instrumentação do Código fonte....20
5.2.2 Instrumentação da Máquina Virtual (JPDA)....21
5.2.3 Instrumentação de Bytecodes....21
5.2.4 Instrumentação baseada em Aspectos....22
5.3 Compressão do rastro e dados coletados....22
6 Análise Dotplot....23
6.1 Origens....24
6.2 Padrões....24
6.3 Squares....25
6.3.1 Diagonals....26
6.3.2 Light Cross....26
6.3.3 Broken Diagonals....27
6.3.4 Reordered Squares....28
6.3.5 Reordered Diagonals....28
7 Conclusão do Capítulo....29
Capítulo 3. Featincode....30
1 Objetivos....30
2 Motivações....30
3 Descrição....31
4 Características....33
4.1 Extração e marcação do rastro....34

4.1.1 Instrumentação do sistema....34
4.1.2 Execução e marcação....35
4.2 Visualização do Modelo de Código Fonte Extraído do Rastro 36
4.2.1 Visualização em árvore....37
4.2.1.1 Pacotes....37
4.2.1.2 Classes 37
4.2.1.3 Métodos 37
4.2.1.4 Objetos....38
4.2.2 Seleção 38
4.2.3 Marcação....38
4.2.3.1 Parcial....38
4.2.3.2 Total....39
4.2.4 Expansão....39
4.2.4.1 Parcial....39
4.2.4.2 Total....39
4.2.5 Filtragem....39
4.2.6 Visualização de Arquivos de código fonte....40
4.2.6.1 Seleção de visualizador Externo....40
4.2.6.2 Seleção do diretório com os arquivos do código fonte....40
4.3 Visualização do Modelo de Características 40
4.3.1 Visualização em árvore....41
4.3.1.1 Características....41
4.3.1.2 Passos de Execução....42
4.3.2 Carregamento....42
4.3.2.1 Extração do Rastro....42
4.3.2.2 Arquivo Externo....42
4.3.3 Seleção 43
4.3.4 Expansão....43
4.3.5 Marcação....44
4.4 Visualização do Rastro de Execução 44
4.4.1 Separação por threads....44
4.4.2 Número de eventos no rastro....45
4.4.3 Participação e Interseção de elementos no rastro 45
4.4.4 Visualização do arquivo bruto com rastro....45
4.5 Visualização de Métricas relacionando elementos do código fonte e do modelo de características....46
4.5.1 Métricas....47
4.5.1.1 Métricas de Cobertura de Características....48
4.5.1.2 Métricas de Cobertura de Elementos do Código Fonte....48
4.5.1.3 Métricas de Similaridade entre Características....49
4.5.1.4 Métricas de Similaridade entre Elementos do Código Fonte....49
4.5.2 Matrizes....50
4.5.3 Escalas de Cores....51
4.5.3.1 Linear x Logarítmica....52
4.5.4 Zoom....53
4.5.5 Seleção e Detalhamento de métricas....53
4.5.6 Ordenação 54
4.5.7 Filtragem 54
4.5.7.1 Filtragem por tipo....55
4.5.7.2 Filtragem por nome....55

4.5.7.3 Filtragem por métrica....56
5 Arquitetura e Detalhes de Implementação....56
5.1 Subsistema de Instrumentação (Trace Extractor)....57
5.1.1 Captura de eventos....57
5.1.2 Marcação do rastro....58
5.2 Subsistema de Compressão (Trace Compressor)....59
5.3 Subsistema de Análise (Trace Analyser)....61
5.3.1 Visualização....61
5.3.1.1 Aplicação....61
5.3.1.2 Visualização do Modelo de Código Fonte....62
5.3.1.3 Visualização do Modelo de Características....62
5.3.1.4 Visualização do Rastro de Execução....62
5.3.1.5 Visualização das Métricas (Matrizes)....62
5.3.2 Processamento....63
5.3.2.1 Carregamento....63
5.3.2.2 Indexação do Rastro....63
5.3.2.3 Cálculo de Métricas....64
5.3.2.4 Estratégia de Cache....65
5.3.3 Modelo....65
5.3.3.1 Modelo do Código Fonte....66
5.3.3.2 Modelo de Características....66
5.3.3.3 Modelo do Rastro de Execução....67
6 Limitações e pontos de melhoria....68
6.1 Cache....68
6.2 Persistência dos índices....68
6.3 Tratamento de Vários rastros....69
6.4 Integração com análise estática....69
6.5 Compressão do Rastro....69
6.6 Otimização no cálculo de algumas métricas....70
7 Resumo do Capítulo....70
Capítulo 4. Metodologia de Análise....71
1 Objetivos....71
2 Motivações....71
3 Informações x Fontes....72
4 Processo de Análise....75
4.1 Definição dos Objetivos da Análise....75
4.2 Definição do Modelo de Características de Interesse....76
4.3 Planejamento dos Roteiros de Execução....77
4.4 Instrumentação do Sistema....79
4.5 Coleta do Rastro....80
4.6 Análises Visuais....81
4.6.1 Análise do Espalhamento das Características pelo Rastro....82
4.6.2 Análise do Significado e Seleção de Threads de Interesse....84
4.6.3 Análise de Elementos do Código relacionados às Características....86
4.6.4 Análise de Similaridade entre Elementos do Código....91
4.6.5 Análise de Similaridade de Implementação entre Características....98
5 Considerações Finais....101
Capítulo 5. Análise do ArgoUML....103
1 Objetivos Gerais....103
2 Descrição do sistema Alvo....103

2.1 Motivações....104
2.2 Características....105
2.3 Breve Descrição da Interface Gráfica do ArgoUML....106
3 Contexto 1: Características Muito Similares....107
3.1 Objetivos específicos....108
3.2 Descrição dos Roteiros de Execução....108
3.2.1 Seqüência prévia para as seqüências 1 a 4:....109
3.2.2 Seqüência 1:....109
3.2.3 Sequência 2:....110
3.2.4 Sequência 3:....110
3.2.5 Sequência 4:....110
3.2.6 Inicialização:....110
3.2.7 Auto-Críticas 1:....110
3.2.8 Auto-Críticas 2:....110
3.3 Análises e Resultados....111
3.3.1 Dados Coletados....111
3.3.2 Análise do Espalhamento das Características pelo Rastro....112
3.3.2.1 Inicialização....112
3.3.2.2 Auto-Críticas....113
3.3.2.3 Características de Interesse....114
3.3.3 Análise do Significado e Seleção de Threads de Interesse....115
3.3.3.1 Análise do rastro de threads com pequeno número de eventos....115
3.3.3.2 Participação dos pacotes no rastro e inferência da semântica....117
3.3.4 Análise de Elementos do Código relacionados às Características....119
3.3.4.1 Análise de Pacotes....119
3.3.4.2 Análise de Classes....125
3.3.4.3 Análise de Métodos....128
3.3.5 Análise de Similaridade entre Elementos do Código....130
3.3.6 Análise de Similaridade de Implementação entre Características....130
3.3.7 Redução do escopo para análise e compreensão do sistema....130
3.4 Discussão....133
4 Contexto 2: Características distintas com pontos em comum....134
4.1 Objetivos específicos....135
4.2 Descrição dos Roteiros de Execução....135
4.2.1 Seqüência Prévia....136
4.2.2 Sequência 1:....137
4.2.3 Sequência 2:....139
4.2.4 Inicialização:....139
4.3 Análises e Resultados....139
4.3.1 Dados Coletados....139
4.3.2 Análise do Espalhamento das Características pelo Rastro....140
4.3.3 Análise do Significado e Seleção de Threads de Interesse....141
4.3.4 Análise de Elementos do Código relacionados às Características....141
4.3.4.1 Análise de Pacotes....142
4.3.4.2 Análise de Classes....144
4.3.4.3 Análise de Métodos....145
4.3.5 Análise de Similaridade de Implementação entre Características....146
4.3.6 Redução do escopo para análise e compreensão do sistema....150
4.3.7 Influência da ordem de execução sobre as análises....151
4.4 Discussão....153

5 Problemas e limitações da abordagem.....153
Capítulo 6. Trabalhos Relacionados.....155
1 Objetivos.....155
2 Outras abordagens para a localização de características.....155
2.1 Salah e outros.....155
2.2 Antoniol e Guéhéneuc.....156
2.3 Zhao e Zhang.....158
2.4 Rohatgi e Hamou-Lhadj.....159
2.5 Eisenberg e Volder.....161
2.6 Singer & Kirkham.....162
2.7 Lukoit & Wilde.....163
2.8 Liu e Marcus.....164
2.9 Greevy e Ducasse.....166
2.10 Eisenbarth e Koschke.....167
Capítulo 7. Conclusão.....169
Referências Bibliográficas.....173

CAPÍTULO 1. INTRODUÇÃO

1 CONTEXTUALIZAÇÃO

As melhores práticas de desenvolvimento de software orientado a objetos têm contribuído para melhoria de qualidade em muitos fatores internos dos artefatos de software. Duplicações tendem a ser minimizadas pelo uso de construções polimórficas presentes nas linguagens de programação e pela aplicação consistente de padrões de projeto. As conseqüências benéficas dessas práticas se estendem para a melhoria da produtividade no desenvolvimento e também na atividades de manutenção do software. Porém, o entendimento do código fonte a partir de seus requisitos e propósitos originais ainda necessita de mecanismos efetivos para prover o mapeamento entre elementos do código fonte e funcionalidades do sistema.

A rastreabilidade entre requisitos e artefatos do código fonte ainda é um grande desafio para os desenvolvedores (EADDY et al, 2008) (ROHATGI; HAMOU-LHADJ; RILLING, 2008). Uma das principais razões está no fato de que uma simples interação do usuário pode ativar o uso de centenas, talvez milhares, de classes e métodos dependendo do tamanho e complexidade do sistema. Este cenário impõe dificuldades para os desenvolvedores que tentam entender o sistema do ponto de vista do usuário. Por exemplo, supondo que o desenvolvedor deseje saber o que ocorre internamente em um sistema quando o usuário ativa um evento através da interface do software. Uma alternativa simples consiste na depuração e navegação pelo código fonte utilizando recursos dos ambientes de desenvolvimento. Entretanto, mesmo para sistemas de porte médio esta tarefa pode se tornar extremamente laboriosa, pois o número de chamadas pode ser muito grande para ser rastreado. Outra solução é analisar os rastros de execução derivados do evento. Esta é a abordagem base de vários trabalhos na literatura (WILDE; SCULLY, 1995), (WONG et al, 1999), (IBRAHIM et al, 2003), (EISENBARTH; KOSCHKE; SIMON, 2003), (ZHAO et al 2004), (EISENBERG; DE VOLDER, 2005), (GREEVY; DUCASSE, 2005), (SALAH et al, 2006), (CORNELISSEN et al, 2007), (EADDY et al,

2008) (ROHATGI; HAMOU-LHADJ; RILLING, 2008).

Segundo um ditado clássico na engenharia de software, para a evolução de programas “a única constante é a mudança” (1ª lei de *Lehman* (LEHMAN et al, 1997) (LEHMAN, 1980)). A razão disto está na necessidade de adaptação contínua dos sistemas para a satisfação de novos requisitos. Por sua vez, as requisições de alteração de um sistema dificilmente são expressas em termos de elementos do código fonte, por exemplo, “adicione a funcionalidade X à classe A” ou “existe um erro no método m1 que deve ser corrigido” (EISENBERG; DE VOLDER, 2005). Dessa forma, o mapeamento precisa ser realizado entre a visão de alto nível do usuário no domínio do problema e a visão do desenvolvedor no domínio da solução. Essa tarefa torna-se especialmente desafiadora quando o desenvolvedor ainda não está familiarizado com o sistema. Neste caso, o desenvolvedor precisa entender o sistema e descobrir tal mapeamento, para só então utilizá-lo como apoio para as atividades que lhe competem. É neste contexto que o presente trabalho estabeleceu suas direções.

2 Exposição do Problema

O tema principal da pesquisa está na localização de características e entendimento de seu espalhamento pelos elementos do código fonte, com o intuito de apoiar atividades de compreensão e manutenção de software (BIGGERSTAFF et al, 1994)(RAJLICH; WILDE, 2002). Diante disto, foram buscadas respostas para as seguintes perguntas:

1. Dada uma característica implementada num sistema, como localizá-la no código fonte?
2. Quais os elementos do código fonte participam na execução de uma característica? Quais os mais relevantes? Quais são específicos e quais são compartilhados entre características diferentes?
3. Qual a similaridade entre características implementadas num sistema quanto aos elementos compartilhados durante a execução?
4. Qual a similaridade entre elementos do código fonte quanto à participação na execução de diferentes características?
5. Há processamento concorrente durante a execução de determinadas

características? Quais os elementos envolvidos?

3 Hipóteses e Proposta

A hipótese deste trabalho é que, permitindo aos desenvolvedores navegar pela estrutura do código fonte e relacioná-la a um modelo com características de interesse, pode-se reduzir o esforço dispendido para a compreensão do sistema, em especial, quando são encontradas as partes do código mais relevantes para uma características e/ou as mais específicas.

A proposta é fundamentada no uso de uma ferramenta de visualização para a análise da interseção entre elementos de um modelo de características e elementos do código fonte. Ao desenvolvedor é oferecida a possibilidade de trabalhar sob as perspectivas do código fonte, das características ou ambas.

A principal contribuição deste trabalho está em mostrar como a visualização de matrizes com características e elementos do código fonte que se expandem e se contraem podem prover informações úteis para entender o espalhamento de características pelo código fonte em sistemas de grande porte orientados a objetos, o que potencialmente poderá reduzir o esforço em atividades de compreensão e manutenção. O suporte inicial dado pela ferramenta cobre sistemas escritos em linguagem Java, mas pode ser igualmente aplicável a outras linguagens orientadas a objetos.

4 Objetivos

O objetivo deste trabalho é mostrar uma abordagem para localização de características e o entendimento do seu espalhamento pelo código fonte. Para isso, uma ferramenta desenvolvida e as técnicas de análise próprias serão apresentadas. Pretende-se com isso apoiar atividades de entendimento de software orientado por objetos, visando reduzir o tempo e o esforço humano em atividades relativas ao ciclo de desenvolvimento. Para isso, o trabalho explora a combinação de técnicas de Análise Dinâmica, Visualização, Análise e Localização de Características.

5 JUSTIFICATIVAS

Vários estudos apontam as atividades de manutenção como responsáveis pela maior parcela no custo total de um software, com percentuais variando de 50% a 90% ((JARZABEK, 2007), pág.1 Seção 1.1). Porém, um dos pré-requisitos para a realização de manutenções é o entendimento do sistema (RAJLICH; WILDE, 2002), sendo esta atividade grande contribuidora nesses percentuais, especialmente ao se lidar com código não familiar (KO et al, 2006). Nesse sentido, um mantenedor precisa compreender tanto conceitos de domínio do problema, para se fazer entender as necessidades do usuário, quanto conceitos no domínio da solução, para poder implementar as mudanças efetivamente. Mas, a lacuna que existe entre os elementos no domínio do problema e os elementos no domínio da solução precisa ser preenchida por conexões que liguem os dois lados. Em geral, estas conexões não tem definições muito claras e muitas vezes estão apenas na cabeça dos desenvolvedores originais do sistema que podem não estar mais acessíveis (KO et al, 2007), ou, em documentação de projeto já desatualizada com as versões mais recentes da implementação. Nestes casos, a única fonte de informação confiável é próprio código fonte (IBRAHIM et al, 2003). A alternativa poderia ser então o estudo abrangente do código fonte. Mas, em sistemas de grande porte os custos de se buscar essa compreensão ampla podem ser proibitivos. A localização de características de interesse de forma precisa e pontual torna-se então um mecanismo de grande valor para facilitar e reduzir o esforço em atividades de compreensão e manutenção do sistema. Conseqüentemente, estes mecanismos potencialmente contribuirão para a redução nos custos para a produção de software.

6 VISÃO GERAL DA DISSERTAÇÃO

O Capítulo 2 apresenta a fundamentação teórica da pesquisa. O Capítulo 3 mostra a ferramenta desenvolvida, descrevendo suas características, arquitetura e alguns detalhes de implementação. O Capítulo 4 mostra a metodologia proposta para o uso da ferramenta, ilustrada com exemplos da aplicação em um sistema real, porém de pequeno/médio porte. O Capítulo 5 descreve um estudo com um sistema real de grande porte realizado para demonstrar a utilização da ferramenta. O

Capítulo 6 mostra alguns trabalhos relacionados encontrados na literatura. O Capítulo 7 apresenta as conclusões da pesquisa discutindo lições aprendidas, pontos fortes e fracos da abordagem e oportunidades de trabalhos futuros.

CAPÍTULO 2. REFERENCIAL TEÓRICO

1 INTRODUÇÃO

Neste capítulo será apresentada uma revisão na literatura englobando os principais temas envolvidos na pesquisa, dentre eles, Análise (Seção 2), Modelagem (Seção 3) e Localização de Características (Seção 4), Análise Dinâmica (Seção 5) e Análise *Dotplot* (Seção 6).

2 ANÁLISE DE CARACTERÍSTICAS

Nesta seção alguns dos principais conceitos relacionados a Análise de Características[1] serão discutidos. Seção 2.1 traz uma breve descrição das origens e motivações para a análise de características. Seção 2.2 define o conceito de característica adotado neste trabalho. Seção 2.3 fala sobre o modelo de características, que é o seu meio de documentação em um sistema, ilustrando com um exemplo.

2.1 ORIGENS E MOTIVAÇÕES

Os conceitos de característica e modelo de características foram introduzidos inicialmente pelo método de Análise de Domínio *FODA (Feature Oriented Domain Analysis*) (KANG et al, 1990), criado no SEI (Software Engineer Institute) (SEI, 2008c). Czarnecki refere-se a esse método como um dos mais maduros e bem documentados entre os métodos disponíveis nas áreas de *Análise e Engenharia de Domínio* ((CZARNECKI; EISENECKER, 2000), pág. 44, Seção 2.8). O método foi criado como uma alternativa aos demais métodos existentes de *Análise e Engenharia de Domínio*, que buscam apoiar o desenvolvimento de sistemas de software voltados para o reúso e criação de famílias de sistemas (ou *Linhas de*

1 Análise de Características: do inglês, *Feature Analysis*.

Produtos).

A Análise de Domínio *FODA* foi descrita originalmente em (KANG et al, 1990), sendo incorporada posteriormente à abordagem *MBSE (Model-Based Software Engineering* (SEI, 2008b)), também criada pelo SEI. Mais recentemente, esta abordagem foi superada pelo *Arcabouço SPL* (*Framework for Software Product Line Practice, Version 5.0* (SEI, 2008a)). Nesta nova abordagem, a Análise de Domínio é utilizada em conjunto com outras técnicas para formar a *Análise de Linhas de Produtos* que promete resultados superiores aos da utilização da Análise de Domínio isoladamente.

De acordo com a abordagem *FODA*, a análise de características tem como finalidade principal a captura do entendimento do usuário final sobre as capacidades gerais de aplicações em um domínio específico (KANG et al, 1990), (SEI, 2008b). Os produtos da análise de características são modelos de características. Esses modelos contêm os *pontos em comum*[2] e as *variabilidades*[3] entre sistemas relacionados, expressos em termos de características.

2.2 Conceito de Característica

Segundo Czarnecki (CZARNECKI; EISENECKER, 2000), uma característica na literatura sobre Engenharia de Domínio pode ser:

1. Uma propriedade do sistema visível para o usuário final (que segue a definição na abordagem original (KANG et al, 1990));
2. Uma propriedade distinta de um conceito (qualquer elemento ou estrutura num domínio de interesse) que seja relevante para um interessado[4] neste conceito.

Neste trabalho, uma definição similar, porém menos abrangente, é adotada. Assim, uma característica é definida como *uma propriedade distinta de uma funcionalidade do sistema, visível, relevante e passível de ser controlada pelo usuário final*. A característica pode ainda envolver outras características por meio de composição.

Há três tipos básicos de características descritos em (KANG et al, 1990): *obrigatórias*, *alternativas* e *opcionais*. Uma característica é obrigatória quando ela

2 Pontos em comum: do inglês, *commonalities*.
3 Variabilidades: do inglês, *variabilities*.
4 Interessado: do inglês, *stakeholder*.

deve existir em todas instâncias do conceito. As características alternativas representam características intercambiáveis, onde uma única alternativa entre elas deve estar presente no conceito. Por fim, as características opcionais podem ou não estar presentes no conceito modelado. Czarnecki propõe uma extensão a esses tipos com a adição de *características-OU*[5] (CZARNECKI; EISENNECKER, 2000). Características-OU representam características intercambiáveis, mas que, ao contrário das *características alternativas,* não necessitam ser exclusivas; assim, uma instância do conceito pode conter um ou mais representantes de um conjunto de características-OU. O termo instância de conceito refere-se a qualquer representante concreto do conceito que esteja em conformidade com as características que o compõem. Por exemplo, se o conceito modelado for um navegador web, programas como Firefox, Internet Explorer e Opera poderiam ser considerados instâncias desse conceito.

Os pontos em comum num modelo de característica são expressos como características obrigatórias. Por sua vez, a variabilidade é expressa através dos demais tipos de características (alternativas, opcionais e característica-OU).

2.3 Modelo de Características

O modelo de características é o meio de documentação das características de um sistema que, segundo a abordagem FODA, é composto de quatro elementos chaves:

1. Diagrama de Características: representa uma decomposição hierárquica de características com indicações sobre os tipos de características envolvidas (obrigatórias, alternativas, opcionais ou características-OU);
2. Definições de Características: descrevem os significados (ou semântica) de todas as características;
3. Regras de Composição de Características: indicam combinações válidas e inválidas de características. Há duas formas básicas: 1. regras de dependência (a presença de uma característica implica na presença de outra característica devido a uma interdependência); 2. regras de exclusão mútua (quais características não podem coexistir numa mesma instância do conceito) ;

5 Características-OU: do inglês, *Or-Features*.

4. Fundamentação das Características[6]: indicam as razões para a escolha ou descarte de uma característica.

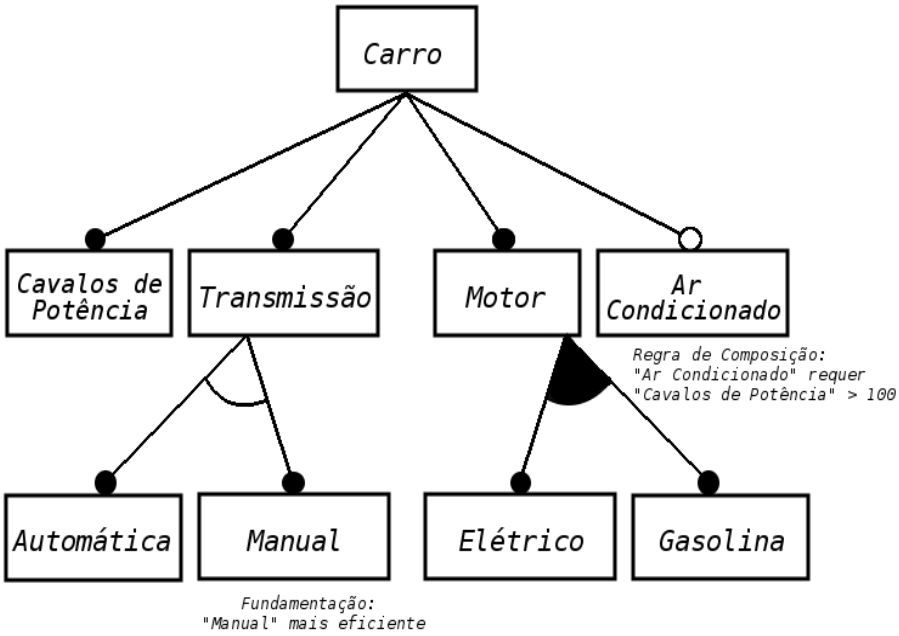


Figura 2.1: Modelo de Características para um Carro, adaptado de (CZARNECKI; EISENECKER, 2000).

A Figura 2.1 ilustra um exemplo simples de *diagrama de características* representando um carro (adaptado de (CZARNECKI; EISENECKER, 2000), páginas 39 e 104). O diagrama mostra três características mandatórias, inerentes a qualquer carro, em conformidade com o modelo: *Cavalos de Potência*, *Transmissão* e *Motor*. A *Transmissão* representa um ponto de variação, podendo ser *Manual* ou *Automática*, representado por características alternativas que são auto-excludentes. No motor há outro ponto de variação, porém não excludentes, indicando que pode haver um *Motor Elétrico*, a *Gasolina* ou ambos (característica-OU). Finalmente, a característica *Ar Condicionado* é opcional, indicando que um carro pode ou não tê-lo. Há uma *regra de composição* indicando que um carro com *Ar Conficionado* requer um valor superior a 100 *Cavalos de Potência* (regra de dependência). Há também a descrição d*a fundamentação das características* indicando que a *Transmissão Manual* é mais eficiente que a *Automática*.

Outros elementos podem compor um modelo de características tais como: *stakeholders* e programas clientes, sistemas exemplo, restrições e regras de

6 Fundamentação das características: do inglês, *features rationale*.

dependência padrão, sítios de disponibilidade[7], sítios de ligação[8] e modos de ligação[9], atributos aberto/fechado e prioridades. Maiores detalhes podem ser encontrados em (CZARNECKI; EISENECKER, 2000).

3 MODELAGEM DE CARACTERÍSTICAS

A atividade de criação de modelos de características é definida na abordagem FODA como Modelagem de Características. A principal finalidade consiste em modelar as propriedades comuns e variáveis de conceitos e suas interdependências. O modelo de característica tem a função de guardar e organizar essas informações de forma coerente. Por ser uma abordagem de modelagem genérica, podendo lidar com qualquer tipo de conceito, essa atividade pode ser combinada a qualquer outra técnica de modelagem, tais como modelagem de casos de uso, modelagem de classes, especificação de funções e procedimentos, e assim por diante.

A seguir serão descritas as etapas necessárias para o processo de modelagem. A identificação das fontes de informação e as estratégias para a definição das características são apresentadas nas Seções 3.1 e 3.2. Seção 3.3 traz os passos deste processo e, finalmente, Seção 3.4 mostra o produto final do processo, o Modelo de Característica.

3.1 FONTES DE INFORMAÇÃO

O primeiro passo na modelagem de características consiste na identificação de suas fontes de informação. Czarnecki sugere as seguintes fontes ((CZARNECKI; EISENECKER, 2000), página 119):

- *Stakeholders* existentes e potenciais;
- Especialistas e literatura do domínio;
- Sistemas existentes;
- Modelos pré-existentes (casos de uso, modelos de objetos e assim por diante)

7 Sítios de disponibilidade: do inglês, *availability sites*.
8 Sítios de ligação: do inglês, *binding sites*.
9 Modos de ligação: do inglês, *binding modes*.

- Modelos criados durante o desenvolvimento (modelos de projeto e implementação).

3.2 ESTRATÉGIAS DE IDENTIFICAÇÃO

Com a identificação das fontes para extração de características, o próximo passo na modelagem é a identificação das características propriamente ditas. Para isso a adoção de estratégias é crucial para o sucesso na identificação. Existem estratégias *top-down* e *bottom-up* para identificação de características. A seguir são enumeradas algumas delas:

- Busca por terminologia de domínio que implique em variabilidade: por exemplo, conta-poupança versus conta-corrente para um sistema financeiro, disposição em cascata versus disposição lado-a-lado num sistema de gerenciamento de janelas, formato MP3 versus formato OGG num sistema de áudio. Qualquer coisa que o usuário deseje controlar em um conceito deve ser considerada uma característica durante a modelagem, incluindo não só características funcionais como também características de implementação;
- Exame de conceitos de domínio em diferentes fontes de variabilidade: buscar nas diferentes fontes de informação (sistemas, *stakeholders*, dentre outros) quais conjuntos de requisitos podem ser postulados para diferentes conceitos de domínio;
- Uso de "*starter sets*" para iniciar a análise: um "starter set" consiste em um conjunto de perspectivas para a modelagem de conceitos e variam conforme o domínio selecionado. Por exemplo, no domínio de algoritmos e considerando a modelagem de tipos abstratos de dados (*Abstract Data Types* ou *ADT*), um possível "*starter set*" incluiria elementos como: atributos, operações, gerenciamento de memória, otimizações e sincronização. Estes elementos representam aspectos importantes a serem considerados no domínio e, dessa forma, fontes valiosas de características para o conceito modelado.
- Busca por características em qualquer ponto do desenvolvimento: como a modelagem de características é aplicável em qualquer nível de abstração (requisitos, arquitetura, projeto, implementação) deve-se estar

atento tanto para a manutenção dos modelos criados quanto para a descoberta de novas características em qualquer etapa do ciclo de desenvolvimento.

- Identificação de mais características do que se pretende implementar: diferentemente de abordagem de modelagem convencional cuja a recomendação é de evitar a modelagem especulativa que não será convertida em implementação inicialmente, na modelagem de características é importante identificar elementos potenciais para dar margem ao crescimento futuro do sistema. Isso permite a definição de interfaces cliente e de configurações mais robustas.

3.3 Processo de Modelagem

O processo de modelagem das características é interativo e executado em ciclos pequenos e rápidos, podendo ser chamado por isso de “micro-ciclo”. Envolve os seguintes passos:

1. Registro de similaridades entre instâncias de um conceito (características comuns);
2. Registro de diferenças entre instâncias (características variáveis);
3. Organização das características em diagramas, incluindo sua classificação (obrigatória, alternativa, opcional, característica-OU);
4. Análise de combinações e interações entre características, buscando por configurações inválidas, dependências ou novas características;
5. Registro de todas as informações adicionais relacionadas às características, tais como, descrições semânticas, *rationales*, *stakeholders* e programas clientes, exemplos de sistemas, restrições, dependências pré-definidas, sítios de disponibilidade, sítios de ligação, modos de disponibilidade, atributos aberto/fechado e prioridades.

3.4 Criação do Modelo de Características

O produto final da modelagem de características é um modelo de características que possua ao menos os elementos chaves citados na Seção 2.3:

diagrama, descrições, *rationale* e regras de composição de características. O diagrama de características em especial possui algumas peculiaridades em sua representação que serão discutidas a seguir.

Um diagrama de características é formado por um conjunto de nós, arestas direcionadas e decorações nas arestas. Os nós e arestas formam uma árvore. As decorações são desenhadas como arcos ligando nós que tem origem em um mesmo nó e formando assim partições com os sub-nós de um nó. A raiz de um diagrama de características representa um conceito, enquanto os nós restantes são as características de um conceito.

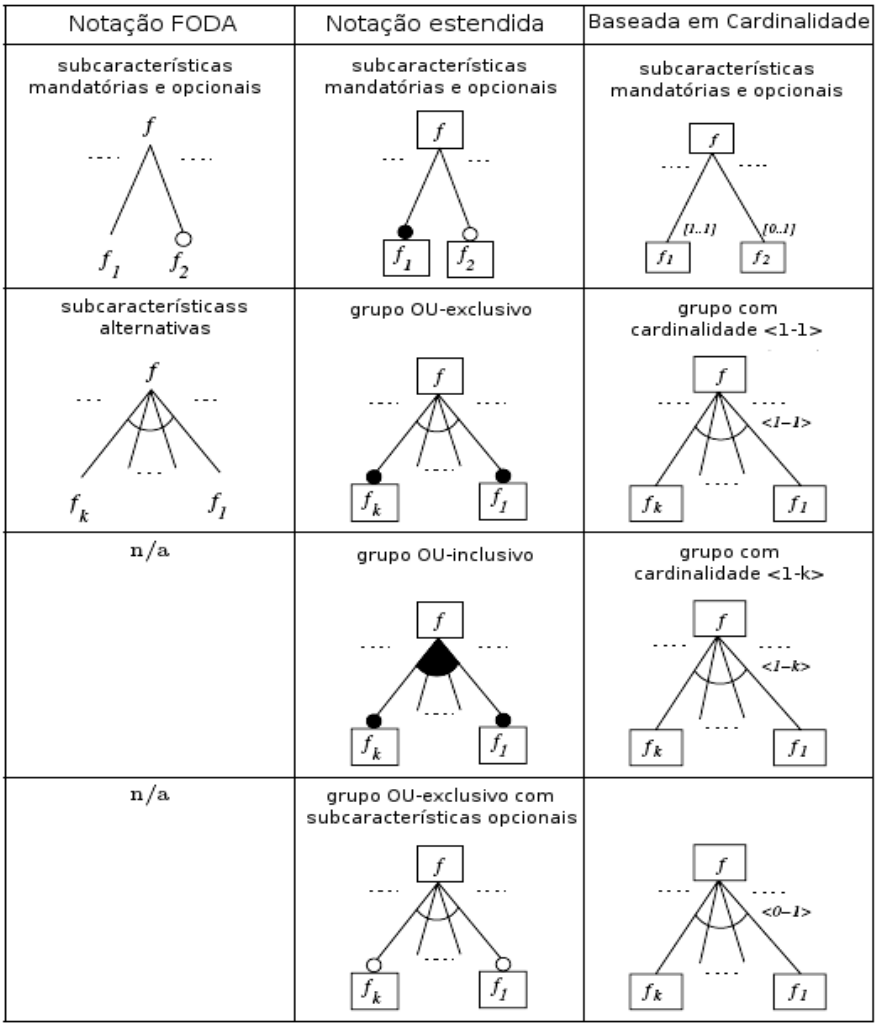


Figura 2.2: Notações para modelagem de características.

Seguindo a notação em (CZARNECKI; EISENECKER, 2000), que estende a notação original em (KANG et al, 1990), características obrigatórias são indicadas com uma aresta e um círculo preenchido em sua extremidade inferior. Uma

característica opcional é indicada também por uma aresta simples, porém com um círculo vazio. Um conjunto de características alternativas é indicado por um arco simples ligando as arestas dessas características em sua extremidade superior. Características-OU são indicadas de forma similar, porém com um arco preenchido. A Figura 2.2 ilustra a notação descrita em contraste com a notação original e com uma notação baseada em cardinalidades, descrita em (CZARNECKI; HELSEN; EISENECKER, 2004). Pode ser notada uma variação para características alternativas na última linha. Esta variação considera que as características alternativas possam ser também opcionais (ou seja, uma configuração contendo uma ou nenhuma das características seria igualmente válida).

A criação dos modelos pode ser apoiada por ferramentas tais como *FeaturePlugin* (ANTKIEWICZ; CZARNECKI, 2004), *CASE-FX* (FORGET; ARNOLD; CHIASSON, 2007), *XFeature (ROHLIK; PASETTI, 2004)* e *CaptainFeature* (BEDNASCH; LANG, 2005).

4 Localização de Características

As funcionalidades de um sistema são capturadas em sua especificação de requisitos e de alguma forma precisam ser traduzidas em código para efetivamente agregar valor ao sistema. Um dos pré-requisitos para lidar com sistemas já desenvolvidos, ou legados, é entender e localizar no código fonte onde as funcionalidades foram implementadas. Porém, a recuperação deste mapeamento entre funcionalidades e código fonte precisa de mecanismos automáticos, especialmente na análise de sistemas desconhecidos. Este processo de localizar partes no código fonte que implementam uma funcionalidade específica é chamado de *localização de características* sendo objeto de estudo de muitos trabalhos na literatura (EISENBARTH; KOSCHKE; SIMON, 2003), (ZHAO et al 2004), (GREEVY; DUCASSE, 2005), (EISENBERG; DE VOLDER, 2005), (ANTONIOL; GUÉHÉNEUC, 2006), (SALAH et al, 2006), (LIU et al, 2007), (POSHYVANYK et al, 2007), (ROHATGI; HAMOU-LHADJ; RILLING, 2008), (EADDY et al, 2008). Termos similares mas que na prática lidam com a mesma questão, também podem ser encontrados, tais como *localização de conceitos* (RAJLICH; WILDE, 2002), *atribuição de conceitos* (BIGGERSTAFF et al, 1994) e *identificação de características*

(ANTONIOL; GUÉHÉNEUC, 2006).

Algumas motivações para o processo de localização de características são apresentadas na Seção 4.1. Os problemas que podem ocorrer são citados na Seção 4.2. Seção 4.3 mostra a explicação de algumas soluções propostas para os problemas apresentados.

4.1 MOTIVAÇÕES

A compreensão é uma atividade essencial para o desenvolvimento e manutenção de software, pois é preciso compreender o sistema antes de efetuar corretamente qualquer mudança (RAJLICH; WILDE, 2002). A compreensão norteia também outras atividades como documentação, visualização, projeto, análise, refatoração e reengenharia. Devido a essa importância, a compreensão de sistemas é um campo de estudo intenso que produziu duas teorias clássicas e complementares para a compreensão de software conhecidas como *top-down* e *bottom-up*. Na abordagem *top-down*, um desenvolvedor obtém a compreensão partindo de hipóteses de alto nível que seriam confirmadas ou rejeitadas conforme as evidências fossem encontradas no código fonte. Por outro lado, na abordagem *bottom-up*, o programador partiria de partes específicas do código e pouco a pouco iria se familiarizando com áreas não conhecidas. Neste contexto, a localização de conceitos (ou características) surge como uma alternativa às abordagens clássicas de compreensão e está centrada no papel dos conceitos no software (RAJLICH; WILDE, 2002).

A localização de conceitos apóia-se na idéia de que à medida que os sistemas crescem em tamanho e complexidade, torna-se inviável compreender todas as partes do sistema. Além disso, as requisições de mudança geralmente são expressas usando conceitos de domínio, o que cria a necessidade de um mapeamento entre um conceito e o código que o implementa. Para se manterem produtivos, os desenvolvedores passam a adotar estratégias de busca apenas por partes do código efetivamente relacionadas à implementação de um conceito.

4.2 PROBLEMAS

Em alguns casos, a localização de características pode ser simples e direta, bastando o apelo à intuição do desenvolvedor (que obviamente está atrelado ao seu conhecimento e experiência com o sistema e com o domínio). Porém, na falha da "intuição", mecanismos mais sistemáticos são necessários. O mais simples consiste no casamento de padrões de texto entre termos do domínio e identificadores no software. Para isso uma ferramenta de busca textual, tal como o utilitário *grep* do *UNIX*, pode ser suficiente. Apesar da ampla aceitação da *técnica grep*, há três situações que apontam suas deficiências: 1. por ser baseada na correspondência entre o nome de um conceito e os identificadores no código, sinônimos e homônimos podem criar problemas; 2. quando os conceitos estão inseridos de forma mais implícita no código; 3. quando o desenvolvedor não tem boas sugestões para os identificadores no código. Por conta dessas dificuldades nas abordagens baseadas na intuição e com a *técnica grep*, foram motivadas as pesquisas em busca de soluções mais robustas para a localização de características.

4.3 SOLUÇÕES

Como discutido na seção anterior, a localização de características baseadas na intuição do desenvolvedor ou em *técnicas grep*, nem sempre são suficientes exigindo abordagens mais sistemáticas. As alternativas existentes combinam geralmente várias técnicas de análise, tais como análise dinâmica, análise estática, análise formal de conceitos, modelos de reflexão[10], análise de características e recuperação de informação.

Uma das primeiras alternativas para a localização de características ficou conhecida como *Software Reconnaissance* proposta por Wilde em (WILDE; SCULLY, 1995). A técnica utiliza análise dinâmica orientada por características do sistema com base em dois roteiros de execução (ou *casos de teste* no trabalho original) para identificar os elementos do código associados a uma funcionalidade. A Figura 2.3 ilustra a aplicação da técnica. Supõe-se que o objetivo seja identificar os elementos na implementação associados à funcionalidade de redirecionamento de chamada (*"call forwarding"*) num sistema de chaveamento telefônico (WILDE, 1994).

10 Modelos de reflexão: do inglês, *Reflexion Models*.

O primeiro diagrama é produzido com base num roteiro de execução que deve exercitar a funcionalidade de *redirecionamento de chamada*. O segundo diagrama resulta da execução do sistema sem exercitar a funcionalidade em questão. Finalmente, através da aplicação da técnica, os dois rastros de execução gerados são analisados e um terceiro diagrama é produzido contendo apenas os componentes exclusivos do *redirecionamento de chamadas*. A técnica é útil para apontar pontos de partida para o entendimento de uma funcionalidade especialmente ao lidar com código não familiar (WILDE; CASEY, 1996). Porém, a técnica pode não ser efetiva em alguns casos: 1. quando há compartilhamento total da implementação entre características aparentemente distintas para o usuário; 2. quando existem dependências entre características que não permitam isolar suas execuções; 3. para características mais genéricas que sempre são executadas pelo sistema, independente do roteiro de execução (WILDE, 1994).

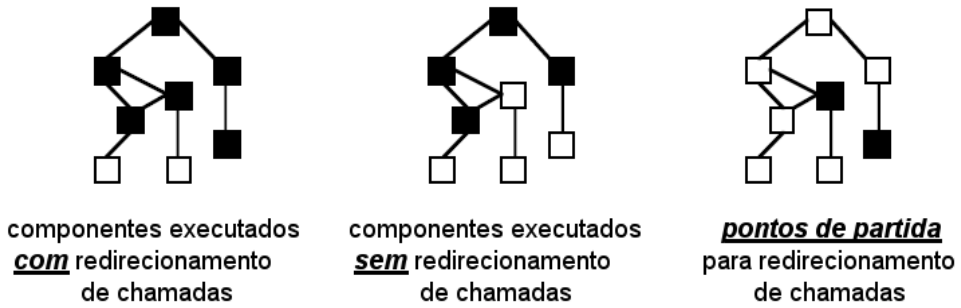


Figura 2.3: Exemplo da execução de *redirecionamento de chamadas* num sistema de chaveamento telefônico extraído de (WILDE, 1994).

Vários estudos de casos (WILDE; CASEY, 1996), (IBRAHIM et al, 2003), (WILDE et al 2003), (JIANG et al, 2006), (JIANG et al, 2007) e técnicas (GEAR et al, 2005) baseadas em *Software Reconnaissance* foram propostos posteriormente. Há também ferramentas derivadas dessa técnica, por exemplo, *Recon* (WILDE, 2007) e *TraceGraph* (LUKOIT et al, 2000). Gear e colegas propõem a técnica *Software Reconnexion*, a qual combina *Reflexion Models* (uma técnica de compreensão *top-down* (MURPHY; NOTKIN; SULLIVAN, 2001)) com *Software Reconnaissance* para a localização de características.

Eisenbarth e colegas (EISENBARTH; KOSCHKE; SIMON, 2001) por sua vez utilizam análise dinâmica e análise formal de conceitos para a localização de características, prometendo ir além da abordagem de Wilde. A abordagem permite

relacionar simultaneamente vários elementos do código (ou componentes de acordo com o trabalho original) às características implementadas. Além disso, mostra pontos em comum e variabilidades tanto entre elementos do código quanto entre características. Numa extensão posterior de seu trabalho, Eisenbarth (EISENBARTH; KOSCHKE; SIMON, 2003) e colegas agregam uma etapa de análise estática para complementar a análise.

Greevy e Ducasse (GREEVY; DUCASSE, 2005) propõem uma técnica para análise das características e elementos do código (ou elementos computacionais, como se refere no trabalho) baseada em análise dinâmica e análise de métricas extraídas dos rastros de execução. Os autores buscam a caracterização desses elementos na execução e fornecem uma terminologia específica para isso.

A proposta de Zhao e colegas (ZHAO et al 2004), ao contrário das anteriores, não se limita à localização de características observáveis e controladas pelo usuário de forma interativa. Para isso, o autor recorre a técnicas de recuperação de informação (RI) e análise estática, denominando sua abordagem de *SNIAFL* (*Static Non-Interactive Approach to Feature Location*). A principal vantagem apontada pelo autor está no melhor suporte para automação, dado que a ligação entre as características e a implementação é feita através de RI ao invés da execução interativa do sistema que pode exigir o planejamento e execução de muitos roteiros de execução do sistema. Dentre as desvantagens estão a definição prévia de descrições para as características, a dependência do uso de identificadores na implementação que sejam consistentes com a descrição de características, e a inflexibilidade na escolha da granularidade mínima do mapeamento, limitada às funções (ou métodos) no sistema.

O uso de RI combinada à análise dinâmica é novamente observado em (LIU et al, 2007). O diferencial da técnica, denominada *SITTIR* (*SIngle Trace and Information Retrieval*), está no uso de um único rastro de execução.

Antoniol e Gueheneuc propõem uma abordagem para comparação e localização de características (ou identificação de características) para sistemas de grande porte, *multi-threaded* e orientados a objetos recorrendo à análise estática e dinâmica, filtragem baseada em conhecimento e *ranking* probabilístico (ANTONIOL; GUÉHÉNEUC, 2005). Em (ANTONIOL; GUÉHÉNEUC, 2006), os autores incorporam à abordagem uma metáfora à epidemiologia. Já em (POSHYVANYK et al, 2007), a técnica de *RI LSI* (*Latent Semantic Index*) é agregada à abordagem,

melhorando os resultados obtidos anteriormente e é denominada *PROMESIR* (*Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval*).

5 Análise Dinâmica

A Análise Dinâmica consiste na análise de propriedades de um programa em execução (BALL, 1999). Enquanto a análise estática examina as informações textuais contidas no código fonte de um programa e deriva propriedades válidas para todas as execuções, a análise dinâmica deriva propriedades válidas para uma ou mais execuções extraídas a partir da execução do programa. Há duas propriedades principais da análise dinâmica: a *Precisão das Informações* coletadas e a *Dependência das Entradas* no programa. A Precisão das Informações diz respeito à capacidade dos mecanismos de extração das informações em ajustar a informação exata a ser coletada conforme as necessidades do analista. A *Dependência das Entradas* refere-se à sensibilidade às variações nas entradas ou interações com o programa durante a execução. Estas propriedades são importantes por permitirem limitar o escopo da análise e relacionar as variações na entrada às mudanças internas no comportamento dos programas.

As seções a seguir mostram como o processo de análise dinâmica ocorre. Seção 5.1 mostra como é feito o planejamento da execução. Seção 5.2 traz diferentes mecanismos de coleta dos dados. Seção 5.3 trata sobre a compressão do rastro e dos dados coletados.

5.1 Planejamento da execução

Em função da natureza da análise dinâmica, sensível às entradas do programa durante a execução, deve-se fazer um planejamento prévio para as execuções. Só assim é possível utilizar essa análise de maneira efetiva e satisfazer os objetivos estabelecidos. Este planejamento envolve a *Definição dos Roteiros de Execução* e a *Definição da Forma de Extração dos Rastros*. Cada um desses pontos será discutido a seguir.

5.1.1 Definição dos roteiros de execução

A definição dos roteiros de execução depende da finalidade da análise e deve especificar os passos a serem seguidos para a execução do programa analisado. No contexto específico de localização de características, os roteiros devem estar ligados às características de interesse e exercitá-las seguindo as recomendações próprias de cada abordagem. Obviamente, é exigido uma familiaridade e conhecimento mínimo sobre a aplicação e seu domínio para que o roteiro possa ser definido de forma consistente. A escolha inteligente dos roteiros de execução é o pré-requisito para o sucesso na localização da característica, dependendo principalmente do bom entendimento dos princípios e do seguimento das diretrizes na abordagem utilizada.

5.1.2 Extração de rastro único x Extração de vários rastros por característica

Quanto ao número de rastros de execução extraídos e tratados, há abordagens que lidam com um único rastro para todas as característica de interesse, por exemplo, como nos trabalho de (LIU et al, 2007), (ROHATGI; HAMOU-LHADJ; RILLING, 2008), (SALAH et al, 2006), enquanto outras abordagens utilizam um rastro para cada característica (WILDE; SCULLY, 1995), (EISENBARTH; KOSCHKE; SIMON, 2003), (EISENBERG; DE VOLDER, 2005). Assim, a escolha depende dos princípios de cada abordagem.

5.2 Mecanismos de Coleta

Toda abordagem baseada em análise dinâmica precisa de alguns mecanismos para a extração dos dados durante a execução do programa. Existem vários métodos que variam de métodos intrusivos no código fonte até métodos transparentes que podem ser habilitados e desabilitados com facilidade. Nesta seção são discutidos alguns deles.

5.2.1 Instrumentação do Código fonte

Embora uma das mais simples e diretas, a instrumentação do código fonte é uma das formas mais intrusivas de preparação do sistema para análise dinâmica.

Em geral, a instrumentação consiste na modificação do código para a geração de informações extras a medida que o sistema executa. Embora seja uma forma simples de se fazer a instrumentação, pode dispender um grande esforço manual, além de poluir o código fonte com elementos extras cuja finalidade seja apenas a instrumentação. A compilação das partes afetadas, e talvez de todo o sistema, pode ser necessária. A implementação dessa abordagem pode ser feita pela simples introdução de chamadas de função ou métodos nos pontos onde o registro dos dados seja desejado. Há ainda ferramentas específicas para essa finalidade, com destaque para os projetos da Fundação Apache para serviços de *logging* que dão suporte a diferentes linguagens com bibliotecas específicas para cada uma delas como *Log4j* (*Java*), *Log4cxx* (*C++*), *Log4net* (*.NET*) e *Log4php* (*PHP*) (APACHE SOFTWARE FOUNDATION - ASF, 2008).

5.2.2 Instrumentação da Máquina Virtual (*JPDA*)

Uma alternativa menos intrusiva para a instrumentação do sistema consiste na utilização de recursos do ambiente de execução. Algumas linguagens que rodam sobre máquinas virtuais dispõem desses recursos naturalmente. Em Java por exemplo, existe a *API JVMTI* (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2008) que pode ser usada para instrumentação dos sistemas sendo executados na máquina virtual. Para isso é necessário a implementação do programa de instrumentação baseado na especificação *JVMTI* e que deve então ser habilitado na máquina virtual. Essa abordagem não exige a modificação do código fonte do sistema alvo.

5.2.3 Instrumentação de Bytecodes

A instrumentação de *bytecodes* é outra alternativa para situações em que o código fonte não esteja disponível ou não possa ser recompilado. Esta alternativa exige um bom conhecimento das instruções nativas da máquina virtual onde o sistema será executado, pois é necessário a modificação dos *bytecodes*. Ferramentas como *Java Wiretap* (FIERZ, 2008) usam esta abordagem.

5.2.4 Instrumentação baseada em Aspectos

Uma das aplicações clássicas citadas na literatura sobre Programação Orientada Aspectos é a instrumentação de sistemas. Esta é uma das possibilidades mais flexíveis. A integração entre o aspecto responsável pela captura das informações e o sistema alvo é feita de forma transparente, podendo ser habilitada e desabilitada com facilidade. O controle sobre as partes analisadas pode ser feitos em nível de instruções, acesso a variáveis, métodos, classes ou pacotes. Várias linguagens já dispõem de bibliotecas para programação de aspectos tais como *AspectJ* (THE ECLIPSE FOUNDATION, 2008) e *AspectWerkz* (BONE; VASSEUR, 2005) para *Java*, *AspectC++* (ASPECTC.ORG, 2008) para *C++* dentre outras (WIKIPEDIA, 2008).

5.3 Compressão do rastro e dados coletados

Um dos maiores desafios na utilização de análise dinâmica consiste em lidar com o grande volume de dados produzidos nos rastros de execução. Em geral, quanto maior o sistema sendo tratado maior é o volume de dados gerado em função do número de elementos e de eventos gerados por eles. As demandas por recursos nesses casos são tanto de processamento quanto de espaço. Dessa forma, alguns trabalhos buscam soluções que tornem escaláveis o tratamento desses dados. Os trabalhos de Hamou-Lhadj e Lethbridge nessa área são de grande importância. Dentre as técnicas propostas por esses autores estão: 1. Técnicas de compressão para simplificar a análise dos rastros, focada especialmente na remoção de eventos redundantes tais como chamadas próprias de laços de execução, chamadas recursivas e seqüências de chamadas que se repetem no rastro (HAMOU-LHADJ; LETHBRIDGE, 2002); 2. Metamodelos para a codificação eficiente dos dados coletados, tais como o *CTF (Compact Trace Format) (HAMOU-LHADJ; LETHBRIDGE, 2004)*; 3. Métricas para medir propriedades dos rastros visando auxiliar a construção de ferramentas de análise do rastro, as quais são divididas em métricas sobre o volume de chamadas no rastro, métricas sobre os componentes do sistema (ou elementos do código) presentes no rastro, métricas sobre unidades de compreensão (sub-árvores distintas formadas a partir do grafo de chamadas do rastro) e métricas para padrões (unidades de compreensão recorrentes no rastro)

(HAMOU-LHADJ; LETHBRIDGE, 2005); 4. Técnicas de sumarização do rastro baseado em métricas do tipo *Fan-in/Fan-out* são calculadas para as chamadas no rastro(HAMOU-LHADJ; LETHBRIDGE, 2006).

6 ANÁLISE DOTPLOT

A análise *Dotplot* consiste em uma técnica de visualização cujo objetivo está em identificar padrões em seqüências de dados (CHURCH; HELFMAN, 1993), (HELFMAN, 1994) e (HELFMAN, 1996). A seqüência de dados é fornecida como entrada e uma matriz é gerada onde as linhas e colunas correspondem a cada um dos itens nessa seqüência. Quando um item em uma linha casa com um item em uma coluna, a célula correspondente é marcada com um ponto (por isso, o nome *Dotplot*). Por ser uma matriz de similaridade, as células na diagonal principal são sempre marcadas, já que as linhas e colunas representam o mesmo item na seqüência. Assim, as células fora da diagonal principal indicam casamento de itens em diferentes posições na seqüência. A Figura 2.4 ilustra a aplicação da técnica, inicialmente sobre uma seqüência de palavras envolvendo a pequena citação *"to be or not to be"* e depois em uma coletânea de textos de *Shakespeare* envolvendo mais de um milhão de palavras. Como será visto na Seção 6.2, vários padrões podem ser identificados nessas matrizes, os quais caracterizam situações singulares (HELFMAN, 1996). Seção 6.1 mostra as origens da análise *Dotplot*.

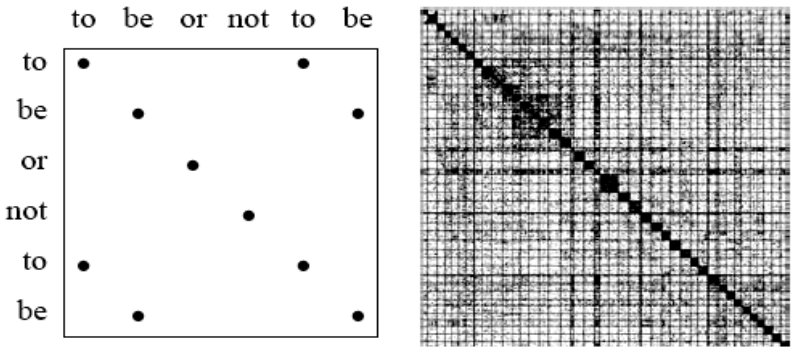


Figura 2.4: a) Exemplo da técnica Dotplot sobre a pequena citação *"to be or not to be"*; b) Aplicação da técnica sobre um coletânea das obras de Shakespeare.

6.1 ORIGENS

A análise *Dotplot* foi proposta por *Church* e *Helfman* em (CHURCH; HELFMAN, 1993) como um técnica de visualização para a análise de similaridade de grandes volumes de dados, tendo sido adaptada da Biologia, onde era empregada na análise de seqüências genéticas. A técnica tira proveito da capacidade humana de reconhecimento de padrões visuais para a identificação de similaridades, a qual é menos sensível a ruídos que abordagens algorítmicas tradicionais (por exemplo, algoritmos usados no casamento de cadeias de caracteres, tais como *longest-common-substrings*, *suffix-trees* ou técnicas de programação dinâmica usadas no utilitário *diff* do UNIX). Em (HELFMAN, 1994) e (HELFMAN, 1996), vários padrões são propostos para auxiliar a interpretação dos gráficos *Dotplot*. As principais aplicações levantadas para a técnica incluem análise de textos (identificação de autores, detecção de plágio, alinhamento de traduções, etc), engenharia de software (identificação de módulos e versões, categorização de subrotinas, identificação de código redundante, etc), recuperação de informação (identificação de registros similares em resultados de consultas), além das tradicionais no âmbito da Biologia (HELFMAN, 1996). Em software, por exemplo, a técnica pode ser empregada para a descoberta de grandes estruturas e para remover e lidar com duplicações indesejadas (CHURCH; HELFMAN, 1993).

6.2 PADRÕES

Os padrões propostos por Helfman (HELFMAN, 1996) serão descritos brevemente nesta seção. As figuras 2.5 a 2.10 ilustram alguns desses padrões. Acima da matriz encontra-se o tipo de seqüência que dá origem ao respectivo padrão.

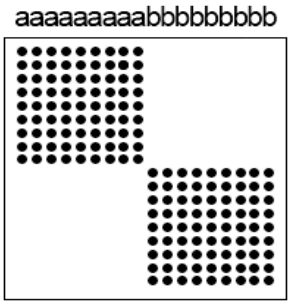


Figura 2.5: Padrão *Square*.

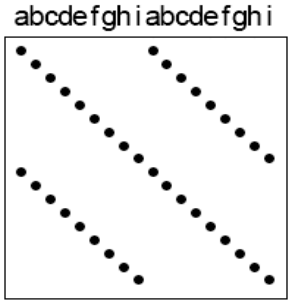


Figura 2.6: Padrão *Diagonals*.

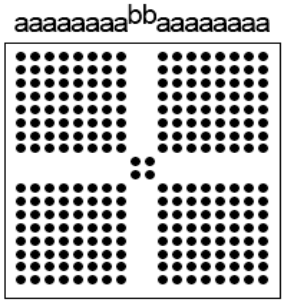


Figura 2.7: Padrão *Light Cross*.

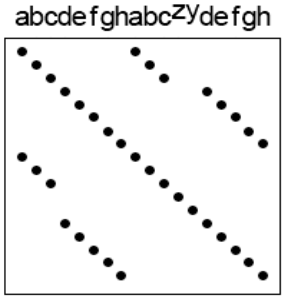


Figura 2.8: Padrão *Broken Diagonals*.

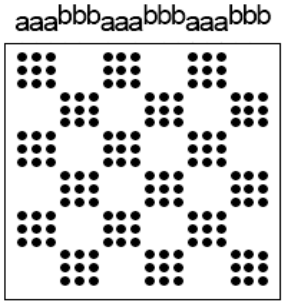


Figura 2.9: Padrão *Reordered Squares*.

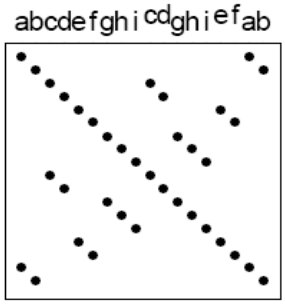


Figura 2.10: Padrão *Reordered Diagonals*.

6.3 SQUARES

Os padrões *Squares* indicam casamentos *não ordenados* na seqüência de itens comparados (Figura 2.5), tais como os ocorridos em documentos com várias palavras similares e em subrotinas de programas com vários símbolos iguais ou similares. O padrão ocorre especialmente na diagonal principal da matriz, onde blocos escuros são formados. A Figura 2.4 exemplifica esse padrão sobre uma

coletânea de documentos. Neste caso, os blocos escuros ao longo da matriz ocorrem principalmente pelo casamento de nomes de personagens numa mesma história e que, no geral, não coincidem em obras diferentes.

6.3.1 Diagonals

O padrão *Diagonals* indica casamentos *ordenados* na seqüência de itens comparados (Figura 2.7), tais como os ocorridos em cópias, versões e traduções de documentos. Neste padrão, diagonais são formadas em paralelo à diagonal principal da matriz. Por exemplo, entre duas versões de um arquivo, a presença de diagonais indica que as versões compartilham um número considerável de palavras mas, ao contrário do padrão *Squares*, essas palavras estão em uma mesma ordem nos arquivos. Como ilustração, duas versões de um sistema chamado `xmh` foram utilizadas na matriz mostrada na Figura 2.11. A matriz foi gerada de forma a apontar quando duas linhas no código fossem idênticas. A presença das diagonais deixa claro que as versões são muito similares e que suas linhas são mantidas na mesma ordem.

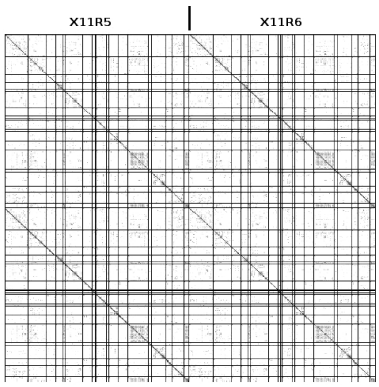


Figura 2.11: Duas versões de `xmh`, um sistema com 20.000 linhas de código C contidas em 53 arquivos.

6.3.2 Light Cross

Em geral, o padrão *Light Cross* indica itens ou seqüências avulsas inseridas em uma seqüência de itens relacionados e não ordenados (Figura 2.7). Em outras palavras, indicam elementos sem relação direta presentes no meio de um grupo de elementos relacionados. O padrão é caracterizado pela presença de uma cruz clara

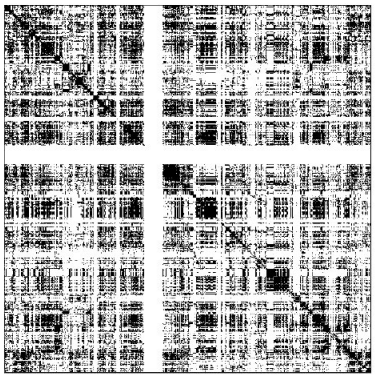


Figura 2.12: 290.000 nomes de arquivos e o padrão *Light Cross* indicando uma seqüência de 15.000 arquivos com nomes únicos.

ao meio de blocos escuros de elementos. A Figura 2.12 ilustra o padrão numa matriz onde a entrada são nomes de arquivos extraídas de um sistema de arquivos. Vários arquivos compartilham nomes similares, porém num dado instante foi gerada uma seqüência de arquivos com nomes únicos que gerou a cruz clara na matriz.

6.3.3 Broken Diagonals

O padrão *Broken Diagonals* pode ser visto como o paralelo do *Light Cross*, porém para itens relacionados em seqüência (Figura 2.8). Em outras palavras, é uma variação do *Diagonals*, onde itens avulsos foram inseridos em uma das seqüências "duplicadas". Neste caso, as diagonais secundárias não mostram um paralelismo uniforme em relação a diagonal principal, o que pode ser observado também nos seus tamanhos distintos. A Figura 2.13 ilustra o padrão identificado na matriz para duas versões do *X Toolkit*. O padrão indica que embora a segunda versão guarde muita similaridade com a primeira pela presença das diagonais

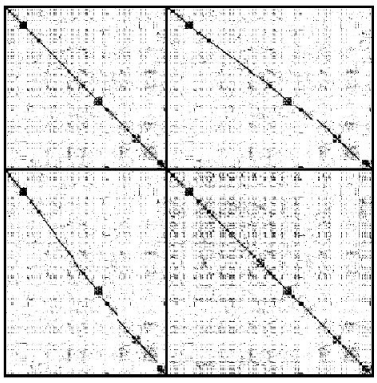


Figura 2.13: Duas versões do *X Toolkit*, envolvendo 66.600 linhas de código C.

secundárias, ela inclui código novo expressos nas “quebras” nessas diagonais.

6.3.4 Reordered Squares

A formação do padrão *Reordered Squares* pode ser imaginada como uma variação do padrão *Square* original, onde as linhas (e colunas) foram reordenadas (Figura 2.9). Assim, ao contrário de blocos escuros ao longo da diagonal, uma textura de blocos fora da diagonal é mostrada espalhada na matriz. A Figura 2.14 ilustra uma situação onde o padrão ocorre e indica inicializações redundantes de objetos que foram reordenadas ao longo de um arquivo de código fonte.

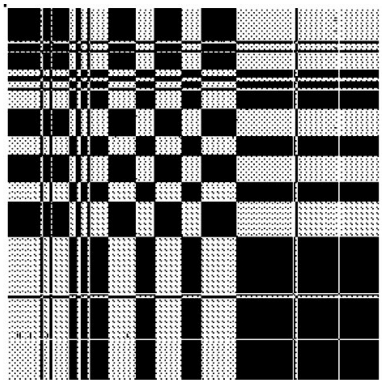


Figura 2.14: Inicializações reordenadas num arquivo com 2500 linhas de código definindo um objeto gráfico.

6.3.5 Reordered Diagonals

De maneira similar ao que ocorre com o padrão *Reordered Squares*, o padrão *Reordered Diagonals* representa uma variação do padrão *Diagonals* com suas linhas (e colunas) trocadas de posição (Figura 2.10). A Figura 2.15 ilustra um exemplo do padrão em duas versões do X Window System são mostradas. Por se tratar de duas versões do mesmo sistema, era esperado a presença de diagonais secundárias como ocorre nos exemplos dos padrões Diagonals e Broken Diagonals. Porém, o que há são fragmentos das diagonais espalhadas pela matriz. Uma explicação para tal ocorrência está na renomeação dos arquivos e a reestruturação dos diretórios de uma versão para a outra. Com isso, embora o conteúdo de muitos arquivos permaneça com similaridades, sua posição nas linhas e colunas da matriz para a segunda versão foi alterada, ocasionando a fragmentação das diagonais

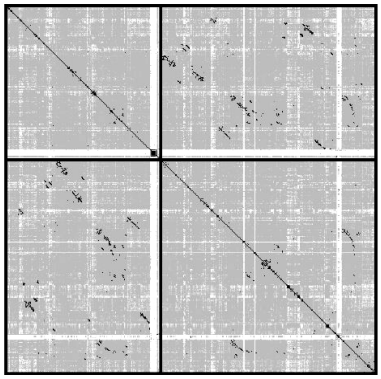


Figura 2.15: Duas versões do X Window System composto por 1.9 milhão de linhas de código.

secundárias.

7 Conclusão do Capítulo

Neste capítulo, algumas das técnicas e conceitos relacionados à Análise, Modelagem e Localização de Características foram discutidos. Embora o foco deste trabalho esteja na Localização, o entendimento de conceitos e técnicas de Análise e Modelagem de Características é um dos pré-requisitos para o uso efetivo da abordagem e da técnica proposta. A definição de um Modelo de Características é o primeiro passo para o uso da abordagem. A Análise Dinâmica também foi discutida. As técnicas desta área são utilizadas na extração de informações durante a execução do sistema sendo analisado. Especificamente, a Análise Dinâmica baseada em aspectos foi utilizada. Por fim, a Análise Dotplot e alguns de seus padrões foram expostos. Esta análise é utilizada, em especial, para a análise de similaridade entre elementos do código fonte e entre características presentes.

CAPÍTULO 3. FEATINCODE

1 OBJETIVOS

Neste capítulo a ferramenta desenvolvida é descrita incluindo as características implementadas, a arquitetura e alguns detalhes da implementação. Ao final do capítulo serão discutidos alguns pontos a serem aprimorados de forma a superar algumas das limitações na implementação corrente.

2 MOTIVAÇÕES

A ferramenta *Featincode-0.1.7* foi desenvolvida com o intuito de apoiar atividades do ciclo de desenvolvimento, como manutenção e compreensão, pela integração de informações semânticas (baseadas num modelo de características do sistema) com informações de implementação (baseadas num modelo do código fonte). Nesse sentido, a ferramenta tenta oferecer aos desenvolvedores um mecanismo de mapeamento mais eficiente entre as informações de alto nível, frequentemente ligadas à linguagem do usuário e requisitos do sistema, e as informações de nível de implementação, relacionadas aos código fonte. Esse mapeamento faz-se mais relevante especialmente em situações onde o código fonte é a única fonte de informações confiáveis sobre o sistema, e as fontes convencionais de informação, tais como desenvolvedores originais e documentação de projeto, encontram-se inacessíveis ou desatualizados em relação às versões mais recentes do sistema. Dentre os benefícios esperados pelo uso da ferramenta estão: visualização e extração de informações de rastros de execução, mapeamento de características para elementos do código fonte, redução no esforço e custo para a localização e compreensão de partes do código associadas a características de interesse.

3 DESCRIÇÃO

A ferramenta *Featincode* foi desenvolvida em Java, sendo composta de três partes principais: um *extrator*, um *compressor* e um *analisador* dos rastros de execução. O extrator combina também programação baseada em *AspectJ*. A Figura 3.1 resume as atividades envolvidas no processo de utilização da ferramenta. A criação do modelo de características e o planejamento dos roteiros de execução são as únicas atividades alheias ao uso dos elementos da ferramenta. A criação do modelo de características pode ser apoiada por ferramentas como *FeaturePlugin* (ANTKIEWICZ; CZARNECKI, 2004). O planejamento dos roteiros de execução dependerá basicamente do conhecimento do usuário sobre o sistema, podendo ser apoiado por manuais, *websites*, notas de versões, listas de discussão e entrevistas com usuários do sistema.

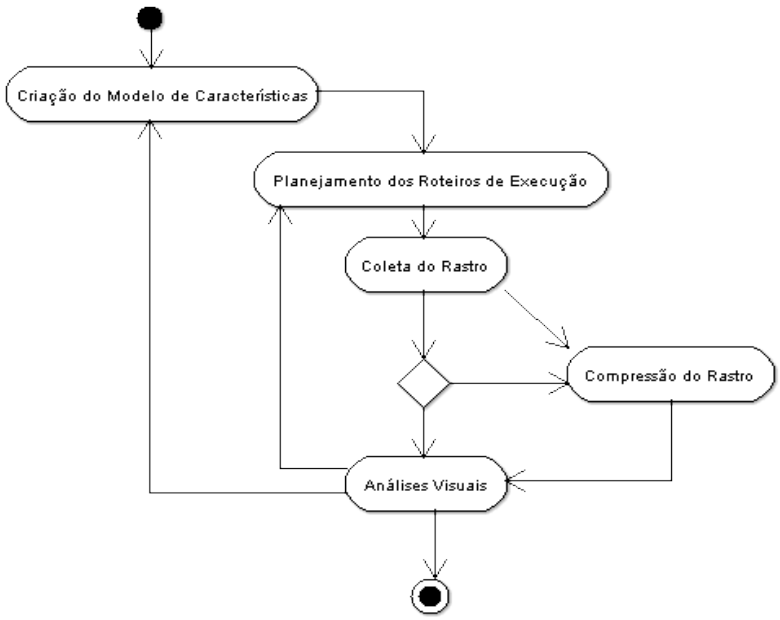


Figura 3.1: Atividades para a utilização da ferramenta Featincode.

A Figura 3.2 mostra a interface de controle do extrator. O sistema alvo executa em paralelo enquanto o usuário marca o início e fim de cada passo de execução responsável por exercitar a característica de interesse. Na figura, a janela à esquerda representa a aplicação *Eye of The Tjger* sendo executada. A janela *Scenario Viewer* mostra os passos de execução previamente definidos pelo usuário. O passo em execução corrente fica em destaque. A janela *Trace Mark Control*

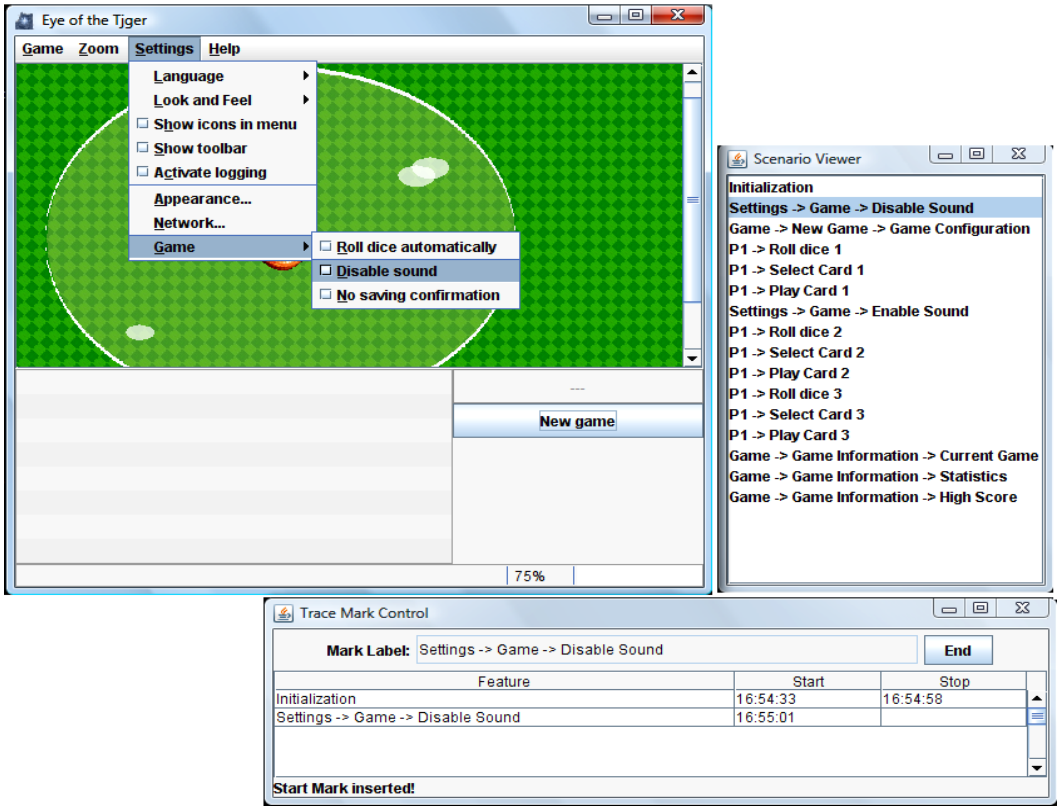


Figura 3.2: Interface de controle do extrator de rastros executando em paralelo com a aplicação *Eye of The Tjger*.

mostra um histórico dos passos já executados e possui o botão de controle para indicar o início e fim de cada passo de execução.

A compressão do rastro é um passo opcional. Há duas motivações para a aplicação da compressão ao rastro: 1. Melhoria do desempenho, dado que a quantidade de dados envolvida no arquivos de rastro é muito grande e o tempo de processamento aumenta proporcionalmente; 2. Redução da influência de eventos redundantes e que muitas vezes não agregam grande valor para a análise, como por exemplo, chamadas recursivas, chamadas em laços e seqüências de chamadas redundantes. A implementação corrente trata os rastros comprimidos como se fossem os rastros originais. A saída do compressor é um rastro compatível e no mesmo formato do rastro original.

A Figura 3.3 mostra a interface gráfica da ferramenta de análise. A janela marcada com *A* mostra todos os pacotes, classes e métodos que aparecem ao menos uma vez nos rastros de alguma *thread*. A janela *B* mostra as características escolhidas pelo usuário e manifestadas nos roteiros de execução. As características podem ser organizadas hierarquicamente, devendo tal organização ser informada

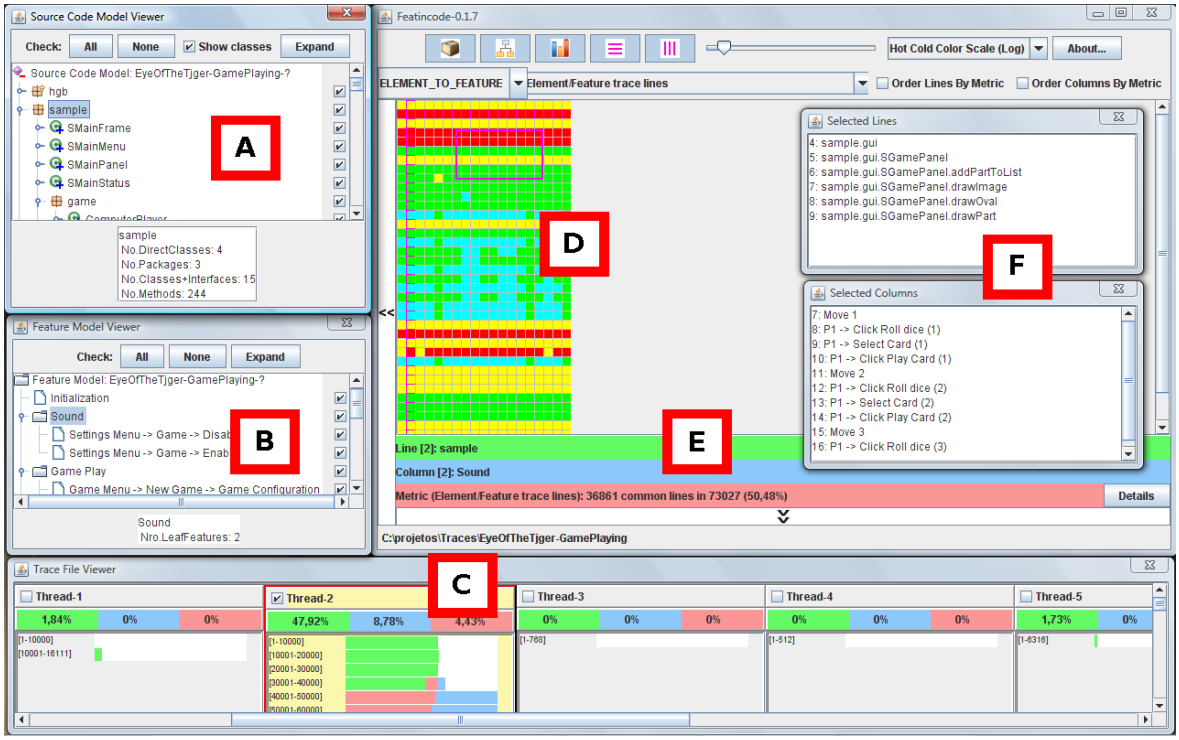


Figura 3.3: Interface principal da ferramenta de análise.

em um arquivo externo. A janela *C* mostra as *threads* que foram iniciadas durante a extração do rastro e os percentuais de participação dos elementos no rastro. A janela *D* mostra matrizes de interseção entre os eventos relacionados às características e os elementos do código fonte de uma *thread* específica.

A matriz na janela D expande e contrai conforme o usuário explora os elementos do código e as características nas janelas A e B, respectivamente. A área E mostra as informações detalhadas da célula selecionada na matriz, informando os elementos do código e as características associados às linhas e colunas, bem como os valores da métrica correspondente. A janela F mostra grupos de linhas e colunas selecionadas na matriz em D, dado que, dependendo do número de elementos sendo exibidos na matriz, torna-se inviável a rotulação de cada linha ou coluna.

4 Características

Nesta seção as características implementadas na ferramenta serão apresentadas e discutidas em detalhes. As configurações e dados prévios necessários (extração e marcação de rastro) para a utilização da ferramenta serão

apresentados na Seção 4.1. As seções seguintes trazem as formas de visualização (ponto importante para a análise) dos vários dados envolvidos no rastro de execução permitidas pela ferramenta: As seções 4.2, 4.3, 4.4, 4.5 e 4.6 apresentam, respectivamente, a Visualização do Modelo de Código Fonte Extraído do Rastro, a Visualização do Modelo de Características, a Visualização do Rastro de Execução e a Visualização de Métricas relacionando elementos do código fonte e do modelo de características.

4.1 Extração e marcação do rastro

Antes da utilização da ferramenta, alguns dados precisam ser extraídos. Inicialmente, um conjunto de roteiros de execução para exercitar as características de interesse devem ser definidos. De posse desses roteiros, o sistema alvo da análise deve então ser executado para que as informações possam ser extraídas. Porém, é necessário que o ambiente de execução seja preparado, o que se resume à *instrumentação do sistema*. Posteriormente, o usuário executa os roteiros de execução, fazendo em paralelo uma *marcação* dos passos para o exercício das características de interesse. As seções a seguir dão mais detalhes sobre estas duas etapas.

4.1.1 Instrumentação do sistema

Em todas as abordagens de análise dinâmica, o sistema alvo passa por um processo de instrumentação para a captura de eventos gerados durante sua execução, seja esta feita diretamente no sistema ou seja pelo uso de algum recurso do ambiente de execução que forneça funcionalidade similar (por exemplo, o uso da plataforma de depuração da máquina virtual em sistemas Java – JPDA).

A captura dos rastros de execução foi implementada usando a abordagem orientada a aspectos por meio de *AspectJ*. Dentre as razões para a escolha da abordagem estão a flexibilidade, facilidade e simplicidade da implementação. Essa abordagem permite capturar os eventos de forma menos intrusiva do que abordagens que exijam a alteração do código fonte. Para isso, basta que a natureza do sistema seja alterada para suportar aspectos e que as bibliotecas utilizadas sejam incluídas no caminho de construção do *AspectJ*. Em função disso, o sistema

deve ser recompilado. Esse pode ser considerado um ponto fraco dessa e de outras abordagens que exigem a recompilação do sistema (na verdade, recompilação através do processo de *weaving* típico da abordagem orientada a aspectos). Uma alternativa para essa questão ainda utilizando a abordagem orientada a aspectos consiste na utilização de *Runtime Weaving*, o que não foi aprofundado nesse trabalho. O uso de *JPDA* também poderia ser uma alternativa viável e que manteria algumas das características da abordagem orientada a aspectos (como a não intrusão no código fonte). Entretanto, a implementação usando *JPDA* é um pouco mais complexa que a abordagem orientada a aspectos.

4.1.2 Execução e marcação

Uma vez feita a instrumentação, a execução do sistema segue o processo normal, exceto pela marcação que deve ser feita pelo usuário, com base nas características de interesse e no roteiro de execução definido. Dentre as razões para o uso do esquema de marcação está a possibilidade da realização de análises de dependência entre as características com base no identificador de objetos instanciados durante a execução, o que, em princípio, só é possível através do uso de um único rastro para os roteiros de execução. Além do mais, o uso de um único rastro pode reduzir o esforço computacional nas análises pela redução no número de eventos processados.

O esquema de marcação consiste na identificação da característica sendo executada e na notificação dos pontos de início e fim da execução dessa característica. As características exercitadas são identificadas de forma seqüencial e única, não sendo permitido sobreposições ou execuções em pontos distintos. Embora seja uma limitação da implementação atual, esse ponto pode ser reconsiderado em implementações futuras, sendo necessário também uma análise de suas implicações. A motivação para esta extensão está na possibilidade de existir intercalação e sobreposição das características durante a execução, especialmente quando há reuso na implementação de características diferentes. Uma alternativa para isso, consiste na combinação e/ou adaptação da abordagem utilizada neste trabalho às abordagens que fazem a marcação do código fonte (SINGER; KIRKHAM, 2008) ou que se baseiam em execuções específicas de métodos para a marcação de características.

A Figura 3.2, página 32, ilustra uma sessão de execução e marcação de características da aplicação *Eye of The Tjger (GRUBER; JORDAN, 2008)*. O passo *"Settings -> Game -> Disable Sound"* está sendo executado para exercitar a característica *"Sound"* da aplicação. Na janela *Trace Mark Control* constam o passo de execução corrente, o botão de controle para a marcação do início e fim dos passos de execução e o histórico de passos de execução realizados (neste caso, *"Initialization" já encerrado e "Settings -> Game -> Disable Sound"* ainda aguardando finalização). A janela *Scenario Viewer* lista todos os passos programados para o roteiro de execução. À medida que o usuário realiza as marcações, o destaque sobre o passo de execução corrente é atualizado.

4.2 Visualização do Modelo de Código Fonte Extraído do Rastro

Vários elementos do código fonte estão envolvidos no rastro de execução, tais como pacotes, classes e métodos. A visualização desses elementos é um ponto importante para a análise. Com base nisso, a janela na Figura 3.4 mostra a interface de visualização para o modelo de código fonte extraído do rastro.

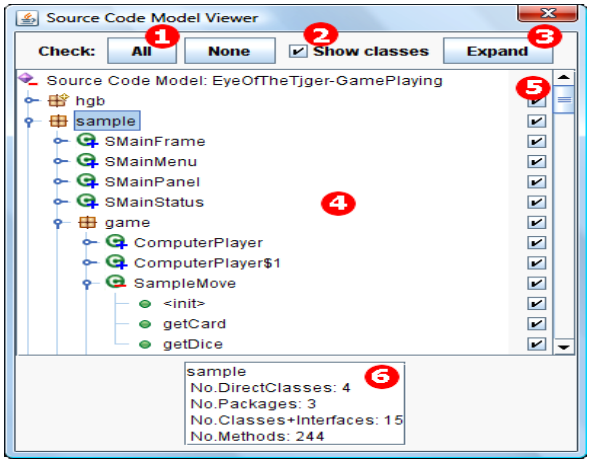


Figura 3.4: Visualizador do modelo de código fonte e controlador dos elementos exibidos na matriz.

Seção 4.2.1 mostra a Visualização em árvore, de pacotes, classes, métodos e objetos. As seções 4.2.2, 4.2.3, 4.2.4 e 4.2.5 trazem as formas de seleção, marcação, expansão e filtragem, respectivamente para auxiliar no processo de visualização. Seção 4.2.6 mostra como é feita a visualização de arquivos de código fonte.

4.2.1 Visualização em árvore

O modelo de código fonte extraído a partir do rastro de execução é apresentado na forma de uma árvore hierárquica, similar às interfaces de exploração de código fonte disponíveis na maioria dos ambientes de desenvolvimento (Figura 3.4, área 4). O objetivo disso foi manter um ambiente familiar para os desenvolvedores que permitisse uma exploração natural e intuitiva dos elementos do código manifestados no rastro. Adicionalmente, foi previsto, embora não implementado, a possibilidade de se contrastar os elementos presentes no rastro com os elementos presentes no código fonte original visando fornecer ao desenvolvedor uma idéia de cobertura da execução do sistema.

4.2.1.1 Pacotes

Os pacotes são visualizados de acordo com a hierarquia sugerida no seu nome qualificado (por exemplo, `sample.game` e `sample.gui`). O intuito foi fornecer informações sumarizadas baseadas no nível de cada pacote. Dessa forma, se o desenvolvedor deseja visualizar a influência de pacotes de mais alto nível nas análises, os pacotes mais próximos à raiz da árvore podem ser priorizados. Uma alternativa de extensão a esse esquema consiste na visualização plana dos pacotes, sem considerar nenhum tipo de composição entre eles.

4.2.1.2 Classes

As classes são atreladas aos pacotes na árvore de visualização com base em seu nome qualificado, similarmente ao que ocorre com pacotes e subpacotes. Uma alternativa ao esquema corrente consiste na visualização exclusiva das classes, sem a informação dos pacotes aos quais elas pertencem.

4.2.1.3 Métodos

Os métodos, por sua vez, estão atrelados às classes que os declaram. Por questões de simplicidade, apenas o nome do método é levado em consideração para distingui-lo dos demais na implementação corrente. A implementação do

analisador e do extrator deve ser estendida para que o desenvolvedor seja capaz de diferenciar o método exato relacionado a um dado evento no rastro em casos onde haja sobrecarga de métodos.

4.2.1.4 Objetos

O último nível possível de visualização seria o nível de objetos. O rastro gerado pela ferramenta permite que essa informação seja extraída. Nesse caso, o objeto poderia estar atrelado à classe da qual foi instanciado, no mesmo nível que os métodos. Porém, a implementação corrente da ferramenta ainda não oferece análises nesse nível, sendo necessária sua extensão.

4.2.2 Seleção

Os elementos selecionados na árvore de visualização definem as informações a serem exibidas em outros pontos da ferramenta de análise. Dentre essas informações destacam-se: estatísticas sobre o elementos presentes no rastro (por exemplo, para pacotes: número de pacotes, classe e métodos envolvidos), as métricas associadas ao elemento quando o mesmo está presente na matriz de visualização e a localização no rastro de execução. A área 6 na Figura 3.4, mostra detalhes sobre o pacote `sample`, selecionado na árvore (área 4).

4.2.3 Marcação

A marcação dos elementos visíveis na árvore define se eles farão parte da matriz de visualização de métricas. Há duas formas de realizar essa marcação: parcial e total.

4.2.3.1 Parcial

Na opção parcial, o usuário define pela interface cada um dos elementos que serão visualizados (área 5 na Figura 3.4).

4.2.3.2 Total

Na opção total, todo o conjunto de elementos mostrados na árvore pode ser marcado para visualização em um único passo (controles na área 1 da Figura 3.4).

4.2.4 Expansão

Visando facilitar a exploração gradual e incremental sobre os elementos da árvore, a expansão dos elementos está disponível nas duas formas discutidas a seguir.

4.2.4.1 Parcial

A expansão parcial segue a interação padrão sobre componentes de interfaces em árvores. O usuário expande os elementos individualmente pelo teclado ou mouse (área 4 da Figura 3.4Erro: Origem da referência não encontrado).

4.2.4.2 Total

Adicionalmente, a expansão pode ser feita de forma total pela ferramenta. Assim, após a seleção de um elemento, o usuário tem a possibilidade de expandir todos os elementos pertencentes a ele em um único passo (controles na área 3 da Figura 3.4Erro: Origem da referência não encontrado).

4.2.5 Filtragem

Com o objetivo de focar em elementos específicos, o usuário tem a possibilidade de filtragem. Na implementação corrente, apenas a exclusão de classes (e conseqüentemente métodos) é possível. Esta opção está disponível através de um controle na interface gráfica (área 2 da Figura 3.4Erro: Origem da referência não encontrado). O uso desta opção implica na visualização apenas da hierarquia de pacotes, excluindo as classes manifestadas no rastro.

4.2.6 Visualização de Arquivos de código fonte

Durante a análise, o desenvolvedor pode desejar examinar o código fonte referente a um dado elemento na visualização (classe ou método). Assim é possível, a partir da árvore do modelo de código ter acesso ao código fonte desse elemento, se este estiver disponível. Para isso duas configurações devem ser feitas pelo usuário como será descrito a seguir.

4.2.6.1 Seleção de visualizador Externo

A visualização do código fonte é realizada através de um visualizador externo, o qual deve ser escolhido pelo usuário. Normalmente um editor simples pode ser utilizado. Basta que o usuário informe à ferramenta a localização do executável em seu sistema operacional, sendo que o mesmo deve aceitar a passagem do arquivo a ser acessado pela linha de comando. Essa alternativa foi utilizada por questões de tempo especialmente. Um editor específico que suportasse a marcação dinâmica de determinadas linhas do código seria o ideal. Porém, a comunicação entre a ferramenta e o editor deveria ser implementada para suportar essa funcionalidade.

4.2.6.2 Seleção do diretório com os arquivos do código fonte

A localização da pasta raiz do código fonte deve ser informada pelo usuário, para que este possa ser acessado pela ferramenta. A implementação corrente suporta apenas um único caminho, mas o suporte e busca por várias localizações está previsto, a exemplo do esquema de busca e carregamento de classes feito pela máquina virtual através do *CLASSPATH*.

4.3 Visualização do Modelo de Características

De maneira similar a visualização do código fonte, a ferramenta de análise permite o carregamento de um modelo de características para auxiliar na análise. A Figura 3.5 mostra a interface de visualização do modelo de características. A área 3 da Figura 3.5, mostra a forma de visualização em árvore, que será descrita na Seção 4.3.1. Seção 4.3.2 traz as formas de carregamento usadas pela ferramenta.

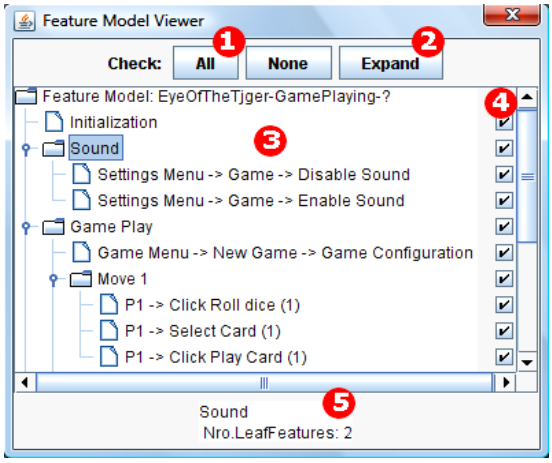


Figura 3.5: Visualizador do modelo de características e passos de execução e controlador dos elementos da matriz.

As seções 4.3.3, 4.3.4, e 4.3.5, mostram, respectivamente, como é feita a seleção, a expansão e a marcação na visualização do modelo de característica.

4.3.1 Visualização em árvore

As características são dispostas em forma de árvore (Figura 3.5, área 3) e dependem do modelo de entrada informado pelo usuário em um arquivo separado (ilustrado a seguir na Seção 4.3.2). Além de buscar refletir a natureza hierárquica das características, essa opção visa também permitir a análise das características em diferentes níveis de granularidade. O suporte a modelos de características é básico, permitindo apenas refletir sua composição. Recursos como a inserção de regras de composição e especificação do tipo de característica (obrigatórias, alternativas e opcionais) ainda não estão disponíveis.

4.3.1.1 Características

As características, em geral, são nós compostos na árvore do modelo de característica. Sua composição inclui outras características ou mesmo os passos de execução para exercitá-las. A Figura 3.5 exibe algumas características como *“Sound”* e *“Game Play”*.

4.3.1.2 Passos de Execução

Os passos de execução, em geral, estão nas folhas da árvore e não contém nenhum sub-item associado. Estão no menor nível de granularidade possível de ser explorado na análise. Além disso, representam os intervalos efetivamente marcados pelo usuário durante a execução do sistema. Alguns exemplos de passos de execução exibidos na Figura 3.5 são *"Settings Menu -> Game -> Disable Sound"* e *"P1 -> Click Roll Dice (1)"*.

4.3.2 Carregamento

O modelo de características usado pela ferramenta de análise tem duas opções de carregamento descritas a seguir.

4.3.2.1 Extração do Rastro

A ferramenta de extração gera um modelo de características automaticamente com base nas características exercitadas pelo usuário. Este modelo porém não considera informações de composição e contém apenas os passos de execução realizados. A Figura 3.6 mostra o modelo criado pela ferramenta de extração com base no rastro. Este modelo pode ser editado pelo usuário e transformado num modelo como o da Figura 3.7 respeitando o esquema descrito a seguir.

```
"Create Class Diagram"
"Insert Interface"
"Insert Class"
"Insert Realization"
"Create Deployment Diagram"
"Insert Node"
"Insert Component"
"Insert Dependency"
```

Figura 3.6: Exemplo de modelo de características gerado automaticamente pela ferramenta de extração.

4.3.2.2 Arquivo Externo

O usuário pode definir o modelo de características em um arquivo externo, podendo definir uma estrutura hierárquica de composição entre as características. O

```
"Draw Diagram"
      "Draw Class Diagram"
            "Create Class Diagram"
            "Insert Interface"
            "Insert Class"
            "Insert Realization"
      "Draw Deployment Diagram"
            "Create Deployment Diagram"
            "Insert Node"
            "Insert Component"
            "Insert Dependency"
```

Figura 3.7: Exemplo de modelo de características editado pelo usuário.

formato de especificação do modelo é textual e bastante simples. Isso pode ser feito também pela edição do modelo gerado pela ferramenta de extração. A Figura 3.7 mostra um exemplo de modelo especificando características relacionadas ao desenho de diagramas UML. As características em negrito representam as características efetivamente exercitadas e marcadas pelo usuário durante a execução do sistema. A composição entre as características é reconhecida com base nas tabulações. O identificador de cada característica deve ser único e estar entre aspas.

4.3.3 Seleção

As características e passos de execução selecionados na árvore de visualização também influenciam outras visualizações na ferramenta de análise, como ocorre para a visualização do modelo de código fonte. Além disso, detalhes sobre o elemento selecionado também podem ser exibidos, como ilustrado para a característica *"Sound"* na área 5 da Figura 3.5.

4.3.4 Expansão

O esquema de expansão na visualização do modelo de características segue lógica similar à visualização do modelo de código descritos na seção anterior. Assim, as duas opções de expansão, manual e automática, também estão disponíveis para a visualização das características (Figura 3.5, áreas 2 e 3).

4.3.5 Marcação

Assim como a expansão, a marcação das características segue a mesma lógica da marcação de elementos do código fonte explorada anteriormente (Figura 3.5, áreas 1 e 4).

4.4 Visualização do Rastro de Execução

A ferramenta provê um esquema de visualização mais amigável e compacto do rastro de execução (Figura 3.8). As áreas marcadas na Figura 3.8 serão descritas nas subseções abaixo: Seção 4.4.1 mostra a separação por *threads*, Seção 4.4.2 mostra como visualizamos o número de eventos nos rastros. Seção 4.4.3 trata sobre a participação e a interseção de elementos no rastro e Seção 4.4.4 trata sobre a visualização do arquivo bruto com o rastro.

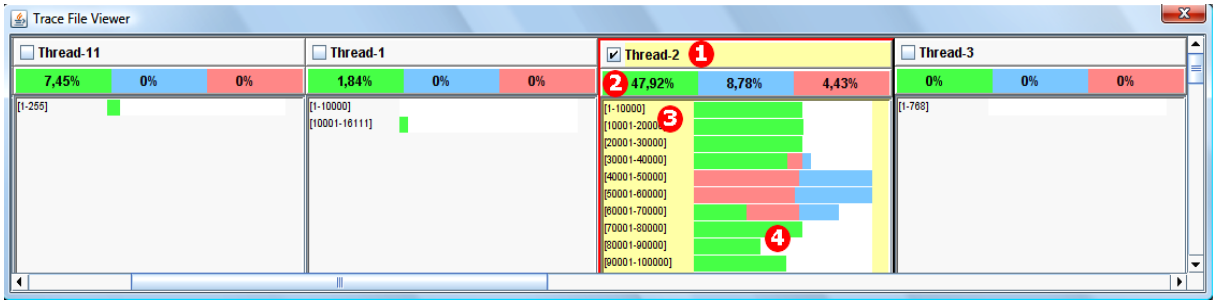


Figura 3.8: Visualizador do rastro de execução mostrando o espalhamento de elementos do modelo de código fonte e do modelo de características.

4.4.1 Separação por *threads*

A identificação das *threads* iniciadas durante a execução permite reconhecer a existência de processamento paralelo no sistema. Cada thread iniciada na execução tem um painel específico rotulado com seu nome (área 1 da Figura 3.8). O controle (*CheckBox*) ao lado do nome da *thread* também pode influenciar as informações exibidas em outras partes da ferramenta, por exemplo, definindo o rastro a ser usado como referência nas matrizes de visualização que usam métricas específicas de uma única *thread*.

4.4.2 Número de eventos no rastro

Como a quantidade de eventos do rastro pode ser excessiva, os eventos são agrupados em intervalos passíveis de configuração pelo usuário (área 3 da Figura 3.8). O total de eventos gerados por cada *thread* pode ser facilmente reconhecido na visualização pela observação do limite superior no último intervalo de eventos.

4.4.3 Participação e Interseção de elementos no rastro

Os intervalos relacionados a elementos do código e às características são destacados na visualização do rastro (área 4 da Figura 3.8). As barras são proporcionais a quantidade de eventos no intervalo relacionados ao elemento. Em princípio, dois elementos são considerados representados pelas cores verde e azul, que devem ser previamente selecionados na ferramenta. Quando há eventos comuns entre os elementos selecionados, a interseção é representada pela barra vermelha. Um sumário em termos percentuais da participação dos elementos e de sua interseção em relação ao total de eventos gerado por cada *thread* é mostrado na área 2 da Figura 3.6.

4.4.4 Visualização do arquivo bruto com rastro

O rastro bruto gerado pela ferramenta de extração pode ser acessado facilmente através da ferramenta de análise. Para isso o usuário deve informar o visualizador externo que será utilizado. O tamanho dos arquivos com rastro de execução pode superar facilmente 1GB sendo recomendado visualizadores especiais para arquivos dessa magnitude. Um exemplo de visualizador simples e funcional para tal finalidade é a aplicação *LTFViewer (Large Text File Viewer)* (CHEN, 2008). Há a pretensão de inserir uma integração maior da ferramenta com esses visualizadores para dar suporte a funções como destaque e posicionamento automático sobre linhas relacionadas a elementos do código ou características selecionados na ferramenta.

4.5 Visualização de Métricas relacionando elementos do código fonte e do modelo de características

A realização das análises visuais é feita com base na extração de métricas do rastro de execução. Para isso, matrizes de visualização são geradas a partir dessas métricas. As matrizes de visualização quantificam relacionamentos entre elementos do código e características. Elas são geradas cruzando dois tipos de informações: 1. as chamadas de métodos entre objetos e classes; 2. as marcações delimitando o início e fim da execução de cada característica. As linhas e colunas representam elementos do modelo de características ou do modelo de código fonte associados à execução do sistema.

A Figura 3.9 ilustra a interface principal para a visualização das métricas. Os controles nas áreas 1 a 3 permitem visualizar/ocultar as janelas de visualização do modelo de código fonte, do modelo de características e do rastro de execução, respectivamente, as quais foram apresentadas nas seções anteriores. Os controles

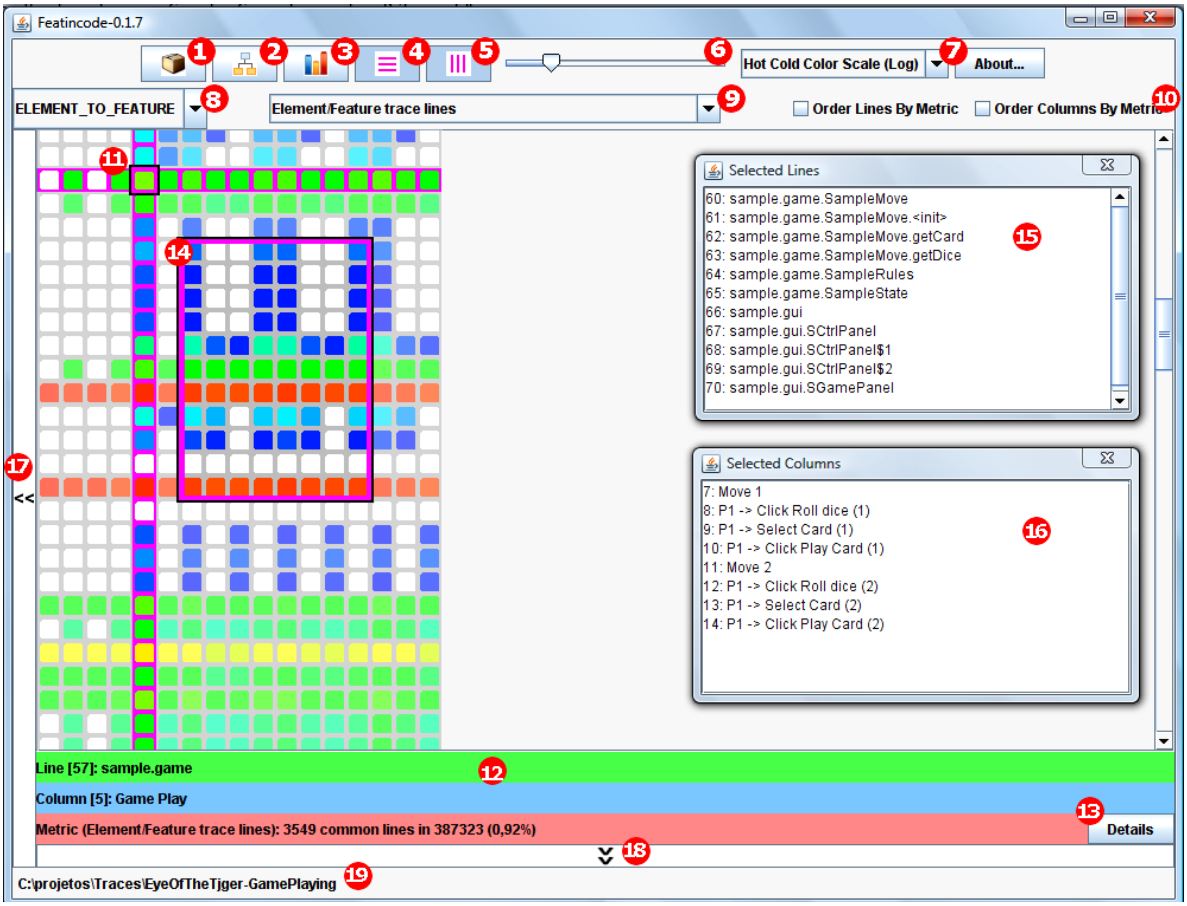


Figura 3.9: Visualizador de métricas.

4 e 5 tem o mesmo efeito sobre os diálogos nas áreas 15 e 16 da figura e que serão discutidos nas próximas subseções. As matrizes são exibidas na área central da janela.

As métricas extraídas do rastro são apresentadas na Seção 4.5.1. Para a visualização das métricas são utilizadas matrizes (Seção 4.5.2), escalas de cores (Seção 4.5.3) e zoom (Seção 4.5.4). Na Seção 4.5.5 é apresentada a forma de seleção e detalhamento de métricas, na Seção 4.5.6 a ordenação, e na Seção 4.5.7 as formas de filtragem.

4.5.1 Métricas

A base de todas as visualizações da ferramentas é formada pelas métricas extraídas do rastro. Essas métricas são dividas em duas categorias: *Métricas de Cobertura* e *Métricas de Similaridade*. Por sua vez, essas métricas podem assumir tanto a perspectiva das *Características* quanto a perspectiva dos *Elementos do Código Fonte*. A Tabela 3.1 sintetiza as métricas consideradas, as quais são detalhadas nas Seções 4.5.1.1, 4.5.1.2, 4.5.1.3, 4.5.1.4. As siglas referentes ao nome das métricas foram mantidas em inglês. A razão disto foi evitar inconsistências ou renomeações quando for feita a tradução deste trabalho para o inglês, no momento da divulgação internacional. As métricas são selecionadas na ferramenta através dos controles nas áreas 8 e 9 mostrados na Figura 3.9.

Tabela 3.1: Métricas de Cobertura e Similaridade para Elementos do Código Fonte e Características.

Tipo	*Perspectiva*	*Alvo*	*Métricas*
Métricas de Cobertura	Características	# Eventos	NSEPF, NSECF, NSEMF.
		# Elementos do Código Fonte	NPPF, NPCF, NPMF.
	Elementos do Código Fonte	# Eventos	NSEFP, NSEFC, NSEFM.
		# Elementos do Código Fonte	$NSPFP_{<d\|s>}$, $NSCFP_{<d\|s>}$, $NSMFP_{<d\|s>}$, $NSMFC_{<d\|s>}$.
Métricas de Similaridade	Características	# Elementos do Código Fonte	NSPF, NSCF, NSMF.
	Elementos do Código Fonte	# Características	NSFP, NSFC, NSFM.

4.5.1.1 Métricas de Cobertura de Características

Para a cobertura das características, a referência pode ser tanto o *Número de Eventos Compartilhados* entre essas características e os elementos do código fonte, quanto o *Número de Elemento do Código Fonte* participantes nessas características. Entende-se por *Eventos* cada chamada presente no rastro de execução, seja ela uma chamada de método, chamado de construtor ou acesso a variáveis de instância. Entende-se por *Elemento do Código Fonte* cada elemento distinto que participa em eventos associados a uma característica, sem se considerar a quantidade de eventos em que o elemento participa. Para essas métricas, a referência de valores está nas características, já que a análise de cobertura está sendo feita sobre elas. Assim, pode ser definido:

NSE<SCE>F[11]: *Número de Eventos Compartilhados entre Elementos do Código Fonte e Características*. Os elementos do código fonte podem ser *Pacotes* (NSEFP), *Classes* (NSEFC) ou *Métodos* (NSEFM).

NP<SCE>F[12]: *Número de Elementos do Código Fonte Participantes na Característica*. Similarmente, os elementos do código fonte pode ser *Pacotes* (NPPF), *Classes* (NPCF), *Métodos* (NPMF).

4.5.1.2 Métricas de Cobertura de Elementos do Código Fonte

A cobertura do rastro pode ser vista como uma espécie de operação inversa à cobertura das características. Neste caso, a referência para as métricas são os elementos do código fonte. A referência para as métricas de cobertura do código pode ser baseada em *Análise Dinâmica* ou baseada na *Análise Estática*. A versão dinâmica leva em conta apenas elementos manifestados no rastro. No caso de se considerar a versão estática, a medida de cobertura daria uma noção da proporção do espalhamento da característica pelo código fonte como um todo, incluindo elementos não manifestados no rastro. Para distinguir ente as duas versões, pode ser introduzido um simples indicador à nomeação. Pode ser definido então:

---

11 NSE<SCE>F: do inglês, *Number of Shared Events between Source Code Elements and Features*. (*Packages* → *NSEPF, Classes* → *NSECF and Methods* → *NSEMMF*).

12 NP<SCE>F: do inglês, *Number of Participant Source Code Elements in Feature*. (*Packages* → *NPPF, Classes* → *NPCF and Methods* → *NPMF*).

NSEF<SCE>[13]: *Número de Eventos Compartilhados entre Características e Elementos do Código Fonte*. Os elementos do código fonte podem ser *Pacotes* (NSEFP), *Classes* (NSEFC) ou *Métodos* (NSEFM).

NS<SCE>F<SCE>$_{<d|s>}$[14] [15]: *Número de Elementos Compartilhados entre Características e Elementos do Código Fonte*. As possibilidades são *Pacotes Compartilhados* (NSPFP$_{<d|s>}$), *Classes Compartilhadas* (NSCFP$_{<d|s>}$) e *Métodos Compartilhados* (NSMFP$_{<d|s>}$ e NSMFC$_{<d|s>}$). O índice *<d|s>* indica quando a métrica é baseada apenas em análise dinâmica (por exemplo, NSPFP$_{<d>}$) ou em análise estática (por exemplo, NSMFC$_{<s>}$).

É interessante notar que a diferença na nomenclatura entre as métricas de cobertura de características e elementos do código fonte é sutil. Porém, a lógica seguida para a nomeação foi colocar o elemento de referência por último. Como a operação é um tipo de projeção, acredita-se que seja a leitura mais natural, tal como se lê A → B para indicar a projeção de A sobre B.

4.5.1.3 Métricas de Similaridade entre Características

A similaridade entre características tem objetivo de compará-las quanto à quantidade de elementos do código fonte compartilhadas entre elas durante a execução. Pode-se definir então:

NS<SCE>F[16]: *Número de Elementos do Código Fonte Compartilhados entre as Características*. Como antes, os elementos do código fonte podem ser Pacotes (NSPF), Classes (NSCF) ou Métodos (NSMF).

4.5.1.4 Métricas de Similaridade entre Elementos do Código Fonte

Como para características, os elementos do código fonte podem ser comparados quanto a participação em características compartilhadas. Para isso, define-se:

---

13 NSEF<SCE>: do inglês, *Number of Shared Events between Features and Source Code Elements*. (*Packages → NSEFP, Classes → NSEFC and Methods → NSEFM*).

14 NS<SCE>F<SCE>: do inglês, *Number of Shared Elements between Features and Source Code Elements*. (*Shared Packages → NSPFP, Shared Classes → NSCFP* e *Shared Methods → NSMFP or NSMFC*).

15 <d|s>: do inglês, *dinamic* e *static*.

16 NS<SCE>F: do inglês, *Number of Shared Source Code Elements between Features*. (*Shared Packages → NSPF, Shared Classes → NSCF* e *Shared Methods → NSMF*).

NSF<SCE>[17]: *Número de Características Compartilhadas entre Elementos do Código Fonte*. Neste caso, as variações para os elementos do código fonte são Pacotes (NSFP), Classes (NSFC) ou Métodos (NSFM).

4.5.2 Matrizes

As matrizes de visualização estão entre as principais formas de apresentação das métricas descritas na Seção 4.5.1. O trabalho de (SANGAL et al 2005) também usa matrizes, porém, para apresentar dependências entre elementos na arquitetura de um sistema e foi uma das inspirações para o uso de matrizes na visualização. As métricas podem ser calculadas para o rastro como um todo ou baseado apenas no rastro gerado por cada *thread*. Isso permite ao analista focar sua atenção em *threads* específicas ou mesmo comparar os eventos entre elas. No momento da escrita deste trabalho, a análise estática ainda não havia sido integrada ao sistema. Assim, as métricas de cobertura do código fonte que necessitam de análise estática ainda não estavam disponíveis ($NSPFP_{<s>}$, $NSCFP_{<s>}$, $NSMFP_{<s>}$ e $NSMFC_{<s>}$). A seleção do tipo de matriz desejada é feita através dos controle na área 8 e 9 da Figura 3.9 seguindo a nomenclatura definida na Seção 4.5.1.

A Figura 3.10 ilustra a configuração das matrizes de visualização de métricas. Na Matriz de Cobertura de Características, os elementos código fonte estão dispostos nas linhas enquanto as características, que neste caso são as referências no cálculo das métricas, estão nas colunas. Na Matriz de Cobertura de Elementos do código, a ordem é invertida, e os elementos do código passam a ser as referências das métricas. Entende-se por referência das métricas o valor base no cálculo, por exemplo, se uma característica está associada a 1000 eventos no rastro, esse é o valor usado como referência na métrica. Assim, considerando um elemento do código que tem 100 eventos em comum com esta característica, a cobertura da característica pelo elemento do código corresponderia ao valor de 100/1000. Por outro lado, se o mesmo elemento participa em 10 mil eventos do rastro, o valor da cobertura do elemento por parte da característica seria 100/10.000. Nas matrizes de Similaridade, os elementos nas linhas e colunas são os mesmos.

17 *NSF<SCE>*: do inglês, *Number of Shared Features between Source Code Elements*. (*Packages* → *NSFP*, *Classes* → *NSFC* e *Methods* → *NSFM*).

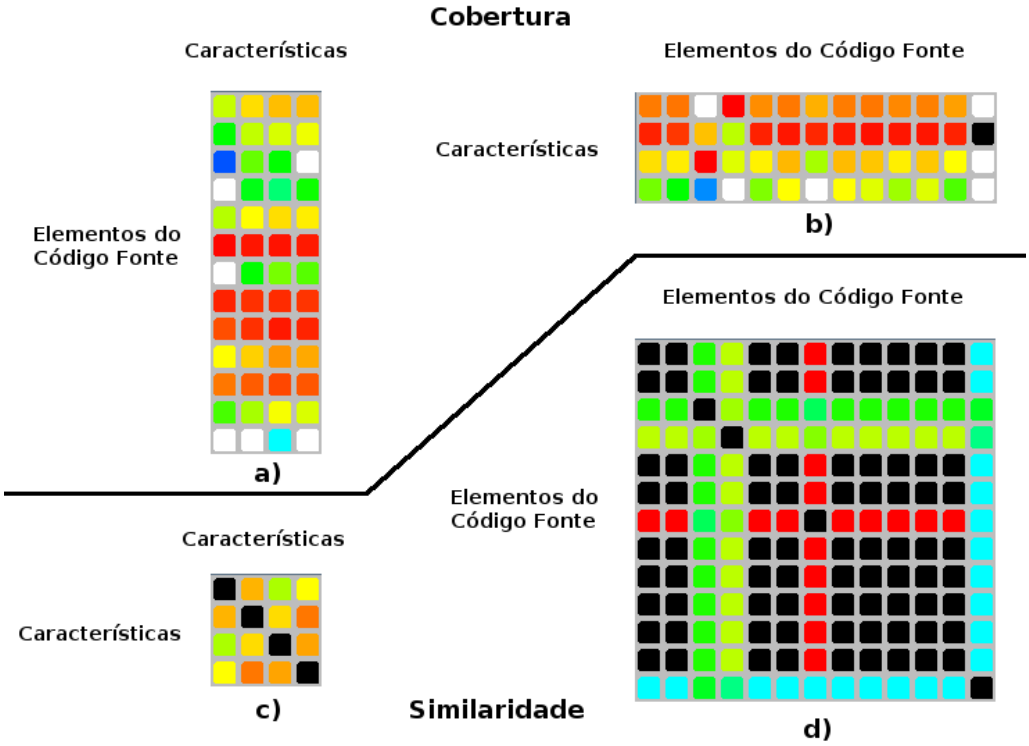


Figura 3.10: Matrizes de visualização de métricas. a) Matriz Cobertura de Características; b) Matriz Cobertura de Elementos do Código Fonte; c) Matriz Similaridade entre Características; d) Matriz Similaridade entre Elementos do Código Fonte.

4.5.3 Escalas de Cores

Com base no valor da métrica calculada, cada célula da matriz é colorida de acordo com o padrão da escala de cores escolhida através do controle na área 7 da Figura 3.9. Um dos objetivos do uso das escalas é o de facilitar a caracterização de elementos com métricas similares. A Figura 3.11 ilustra os mapas de cores utilizados na coloração das células das matrizes, juntamente com os valores correspondentes aos intervalos.

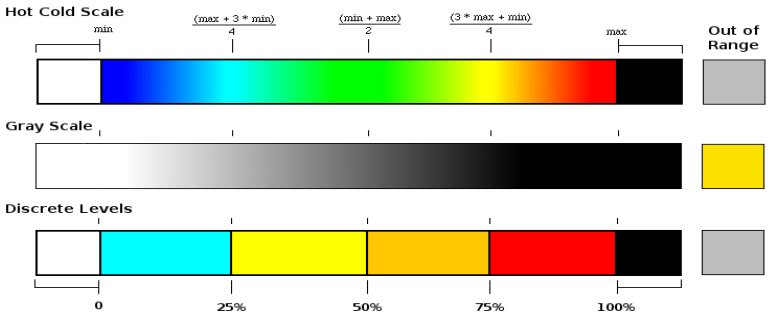


Figura 3.11: Mapas de cores para as escalas Hot Cold, Gray e Discrete Levels.

As escalas *Hot Cold* usam um intervalo contínuo no espaço RGB partindo do azul até o vermelho, exceto pelos extremos inferior e superior, aos quais é atribuída a cor branca e preta, respectivamente. A idéia de diferenciar os extremos foi deixar claro quando uma métrica correspondesse a um desses valores. A escala de cinza (*Gray Scale)* varia em tons de cinza partindo do branco para o valor mínimo e indo até o preto para o valor máximo, também de forma contínua no espaço RGB. Finalmente, a escala discreta (*Discrete Levels*) define quatro intervalos de coloração, mais as cores para os extremos. Todas as três escalas possuem uma cor neutra para indicar quando um valor de métrica está fora da escala ou não pode ser exibido (*"Out of Range"*). Isso ocorre, por exemplo, quando um dos itens indicados na linha ou coluna é a referência para a métrica e, embora participe em algum ponto no rastro, não participe na execução de uma *thread* específica.

4.5.3.1 Linear x Logarítmica

O uso da escala de cor com o valor direto da métrica é a forma normal de coloração das células. Isso produz uma variação linear dos valores sobre a escala. Porém, algumas métricas podem produzir valores muito distantes o que pode levar a interpretações enganosas sobre a posição dos elementos na escala. Por exemplo, suponha o uso da métrica NSECF e uma característica envolvendo 500 mil eventos (*valor referência*). Considere a classe A que participa em 100 eventos, a classe B que participa em 1000 eventos e, finalmente a classe C que participa em 100 mil eventos dessa característica (*valores absolutos*). Tomando em valores percentuais, teria-se 0.02% para A, 0.2% para B e 20% para C. Embora a diferença de participação nas características seja grande e claramente perceptível em valores numéricos absolutos, a coloração das células poderia ser muito próxima usando uma escala linear o que poderia induzir a idéia errônea de que a participação dessas classes seja similar. Na escala com níveis discretos (*Discrete Levels*) todas teriam a mesma coloração por estarem abaixo de 25%. Por outro lado, aplicando uma função logarítmica ($log_{10}valor_absoluto$ / $log_{10}valor_referencia$), os mesmos valores citados passariam para 35% para a classe A, 52% para a classe B e 88% para a classe C. Isso mostra que para métricas com valores tão distantes como os citados, a aplicação da função logarítmica sobre as métricas pode fornecer uma distribuição melhor e mais discriminante nas escalas de cores.

4.5.4 *Zoom*

A opção de *zoom* permite redimensionar a matriz com o objetivo de acomodar na tela um número maior de linhas e colunas e lidar com a questão da escalabilidade na visualização. Para isso, a altura e largura de cada célula é alterada. Dessa forma, mesmo na presença de um número muito grande de elementos ainda é possível obter uma visão geral dos elementos presentes na matriz. O controle na área 6 da Figura 3.9 permite ajustar o nível de Zoom das matrizes, redimensionando as células.

4.5.5 Seleção e Detalhamento de métricas

A métrica correspondente a uma linha e coluna específica pode ser detalhada pela seleção individual da respectiva célula na matriz. O detalhamento ocorre em dois níveis, no primeiro são exibidos os elementos referentes a linha e coluna selecionada, além dos detalhes sobre a métrica (nome e valores). No segundo nível, elementos envolvidos na métrica calculada são listados.

A área 11 na Figura 3.9 mostra uma célula selecionada. Na área 12 pode ser visto o primeiro nível de detalhamento que mostra que a célula está na linha 57 da matriz (correspondendo ao pacote `sample.game)`, na coluna 5 (correspondendo à característica *"Game Play")* e que este pacote está presente em 3.549 linhas das 387.323 associadas a esta característica (ou seja, 0,92% das linhas associadas à característica). Como a escala utilizada foi a "Hot Cold (Log)", a coloração da célula é obtida após a aplicação da função logarítmica sobre a métrica, neste caso log (3.549) / log (387.323) * 100 = 63,53 %. Esse novo valor posiciona a métrica no intervalo entre 50 e 75 % da escala Hot Cold, cujo mapeamento implica na tonalidade verde observada na célula selecionada da Figura 3.9 (área 11).

A área 13 na Figura 3.9 mostra o botão de acesso ao segundo nível de detalhamento da célula selecionada. Para as métricas que fazem a interseção entre linhas, o detalhamento das linhas em comum pode ser visto diretamente no Visualizador de Rastro. Para as demais métricas, o controle dá acesso aos itens em comum/distintos entre os elementos selecionados. A Figura 3.12 mostra a listagem apresentada para o pacote `sample.game` e *"Game Play"* usando a métrica *NPCF* no escopo da *Thread-2*.

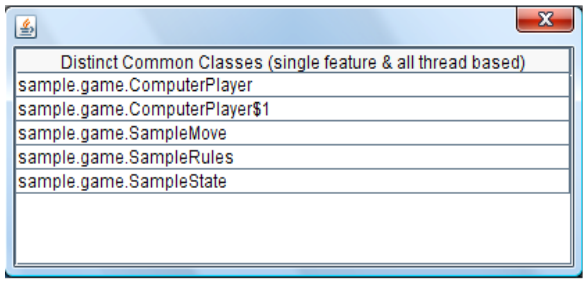


Figura 3.12: Detalhes para a célula selecionada correspondente ao pacote `sample.game` e a característica "*Game Play*".

Além da seleção individual, também é possível selecionar grupos de linhas e colunas (área 14 na Figura 3.9), porém sem a opção de detalhamento de métricas. Neste caso, os elementos correspondentes às linhas e às colunas são exibidos nas janelas mostradas nas áreas 15 e 16 da Figura 3.9. Esta opção foi inserida para permitir que o usuário consiga distinguir os elementos associados às linhas e colunas da matriz. Essa foi a alternativa para a rotulação das linhas e colunas que poderia ser inviável dependendo do número de elementos sendo exibidos na matriz.

4.5.6 Ordenação

Há a opção de reordenar as linhas e colunas da matriz de visualização (área 10 da Figura 3.9). A ordenação padrão na implementação atual é baseada na coloração das células e reflete o intervalo na escala de cores, onde a métrica se encaixa. Outras opções de ordenação são possíveis sendo preciso para isso extensões mínimas na ferramenta contendo novos algoritmos para a ordenação. Essa funcionalidade visa permitir um reconhecimento e agrupamento rápido dos elementos com métricas similares.

4.5.7 Filtragem

Com o intuito de manter o foco sobre um conjunto de elementos específicos a ferramenta possibilita três formas de filtragem: por tipo; por nome; por métrica. Essa filtragem pode ser feita em elementos nas linhas e colunas de maneira independente e simultânea. A Figura 3.13 ilustra a janela de configuração de filtros para as linhas da matriz. As colunas tem uma janela de configuração similar. É importante destacar que essa filtragem é aplicada apenas às linhas e colunas da matriz de visualização e

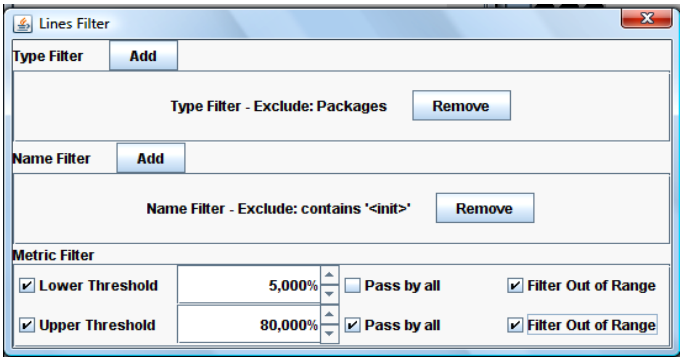


Figura 3.13: Janela de configuração dos filtros para as linhas da matriz.

não influencia as demais partes como os visualizadores do modelo do código ou do modelo de características. Os controles que dão acesso aos filtros estão localizados nas áreas 17 e 18 da Figura 3.9.

4.5.7.1 Filtragem por tipo

Essa filtragem é feita com base no tipo de elemento envolvido. Os tipos suportados no momento são pacotes, classes, métodos ou características. A Figura 3.13 ilustra a configuração para excluir pacotes das linhas da matriz.

4.5.7.2 Filtragem por nome

A filtragem por nome baseia-se no identificador do elemento consistindo no nome qualificado para pacotes, classes e métodos ou no nome para as características. Estão disponíveis os seguintes "operadores" para a aplicação do filtro: *starts with, ends with, contains, equals*. O usuário define o termo e o operador a ser usado sobre ele. Por exemplo, supondo que fosse desejado a remoção de todos os elementos correspondentes a construtores e sabendo que o nome qualificado de um construtor segue a forma "*nome_pacote.nome_classe.<init>*", o uso do termo "*<init>*" e dos operadores *contains* ou *ends with* resultaria na exclusão de todos os construtores presentes na matriz de visualização corrente. A Figura 3.13 ilustra a configuração que permite a exclusão dos construtores com o operador *contains*.

4.5.7.3 Filtragem por métrica

A filtragem por métrica permite definir limiares mínimos e máximos para que um dado elemento seja exibido na matriz. Esse filtro pode ser configurado para que o elemento seja excluído quando uma de suas células correspondentes esteja fora dos limiares (marcando a opção “Pass by all”) ou apenas quando todas as suas células estejam fora dos limiares. Na Figura 3.13, o limiar inferior está definido em 5%, de forma que linhas com ao menos uma métrica acima desse limiar sejam mantidas na matriz. O limiar superior por sua vez está definido em 80% e, neste caso, todas as células devem estar abaixo desse limiar para que a linha seja mantida na matriz. A opção “Out of Range” é habilitada para permitir a exclusão de linhas contendo células com métricas fora da faixa permitida.

5 Arquitetura e Detalhes de Implementação

A Figura 3.14 dá uma visão geral da arquitetura da ferramenta *Featincode*. Como dito anteriormente, há três elementos principais na ferramenta: um

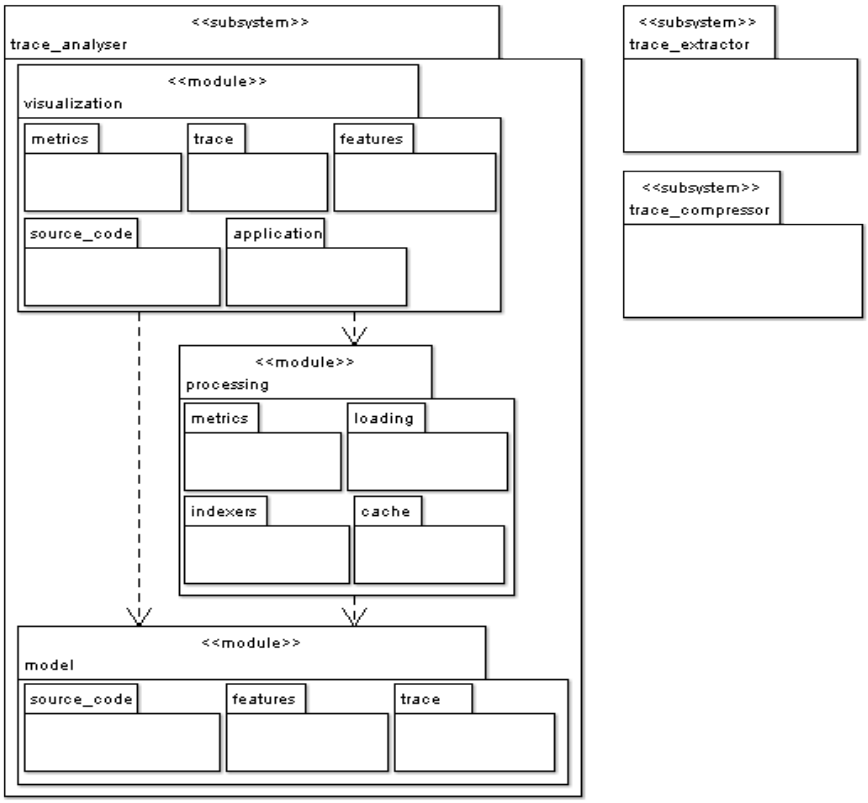


Figura 3.14: Visão Geral da Arquitetura do Featincode.

subsistema de instrumentação, um subsistema de compressão e um subsistema de análise. A troca de informação entre os três subsistemas é feita através do rastro de execução gerado. As seções 5.1, 5.2 e 5.3, respectivamente, darão mais detalhes sobre cada um desses subsistemas.

5.1 Subsistema de Instrumentação (Trace Extractor)

O rastro de execução é gerado pelo subsistema de instrumentação. O usuário precisa apenas especificar um diretório em seu sistema de arquivos para conter os resultados da captura.

5.1.1 Captura de eventos

Os eventos são capturados usando recursos de *AspectJ*. A Figura 3.15 resume o código responsável por interceptar os eventos gerados durante a execução do sistema alvo.

```
public aspect Monitor {
// POINTCUTS
	pointcut alocacaoDinamica(Object objeto) :
		initialization(*.new (..))
		&& target(objeto)
		&& !within(traceextractor..*);

	pointcut execucaoDeMetodo() :
		execution(* *(..))
		&& !within(traceextractor..*);
// ADVICES
	// Registra execução de métodos
	before() :
		execucaoDeMetodo() {
		tracer().registerEvent(thisJoinPoint, TraceEventType.METHOD_ENTRY);
	}

	after() :
		execucaoDeMetodo() {
		tracer().registerEvent(thisJoinPoint, TraceEventType.METHOD_EXIT);
	}

	// Registra criação de objetos
	before(Object objeto) :
		alocacaoDinamica(objeto) {
		tracer().registerEvent(thisJoinPoint, TraceEventType.CONSTRUCTOR_ENTRY);
	}

	after(Object objeto) :
		alocacaoDinamica(objeto) {
		tracer().registerEvent(thisJoinPoint, TraceEventType.CONSTRUCTOR_EXIT);
	}
	// ...
}
```

Figura 3.15: Parte do código responsável pela captura dos eventos no sistema alvo sendo executado.

A cada nova *thread* iniciada na execução, um subdiretório é criado com o nome da *thread*. Nele um arquivo *"data.trace"* é criado para guardar eventos gerados pela thread. Cada evento é adicionado a uma nova linha do arquivo e contém informações sobre o nome completo do método sendo executado (incluindo nome qualificado do pacote e classe) e o instante de entrada no método. Essas informações são extraídas com base na API de *AspectJ* e na API padrão da plataforma Java. Embora não sejam processados pela versão corrente mas visando extensões futuras na ferramenta, o rastro contém também informações sobre o nível de aninhamento do método na pilha de chamadas e um código de identificação do objeto ou classe sendo chamada. A Figura 3.16 dá um exemplo do formato de arquivo de rastro gerado pelas *threads*. Usando a vírgula (",") como separador, os campos do arquivo são: nome qualificado da classe, nome do método, nível de aninhamento e o *timestamp* (representado por um *long*).

```
sample.SMainFrame,main,1,-1,1214381254109
sample.SMainFrame,<init>,2,1,1214381254109
hgb.lib.internal.IntBooleanStringMap,<init>,3,2,1214381254157
hgb.lib.HGBaseSettings,fromFile,3,-1,1214381254157
hgb.lib.internal.IntBooleanStringMap,fromFile,4,2,1214381254157
(...)
```

Figura 3.16: Exemplo de trecho de arquivo "data.trace" com rastro de execução gerado por uma thread.

5.1.2 Marcação do rastro

Durante a execução do sistema, o usuário deve marcar o início e fim da execução da característica, ou mais precisamente dos passos de execução que a exercitam. Essa marcação é feita em paralelo com a interação do sistema alvo. Com base no roteiro de execução previamente definido, um arquivo texto, como ilustrado na Figura 3.17, contendo a seqüência de passos de execução a serem realizados deve ser criado e sua localização informada à ferramenta. O arquivo é carregado pela ferramenta e utilizado para guiar o usuário durante a execução. Caso os eventos de inicialização do sistema devam ser capturados, o arquivo deve incluir esse passo e a ferramenta deve ser informada no início da execução.

```
Initialization
Settings -> Game -> Disable Sound
Game -> New Game -> Game Configuration
P1 -> Click Roll dice (1)
P1 -> Select Card (1
P1 -> Click Play Card (1)
(...)
```

Figura 3.17: Exemplo de trecho de arquivo com passos para um roteiro de execução definido previamente.

Como resultado da marcação feita pelo usuário, um segundo arquivo é gerado ao fim da execução do sistema (*"trace.mark.txt"*). Este arquivo contém a seqüência de execução realizada pelo usuário com a adição dos instantes de início e fim de cada passo de execução (Figura 3.18).

```
#START,Initialization,1214381252141
#END,Initialization,1214381263317
#START,Settings Menu -> Game -> Disable Sound,1214381266536
#END,Settings Menu -> Game -> Disable Sound,1214381273161
#START,Game Menu -> New Game -> Game Configuration,1214381274552
#END,Game Menu -> New Game -> Game Configuration,1214381284150
#START,P1 -> Click Roll dice (1),1214381285025
#END,P1 -> Click Roll dice (1),1214381287901
(...)
```

Figura 3.18: Exemplo de trecho de arquivo contendo as marcações dos passos de execução.

5.2 Subsistema de Compressão (Trace Compressor)

O processo de compressão foi baseado em (HAMOU-LHADJ; LETHBRIDGE, 2002) onde algoritmos para a compressão de rastros de execução são propostos. Enquanto a remoção de eventos redundantes derivados de laços de execução e chamadas recursivas é feita de forma direta, pela simples verificação entre dois eventos sucessivos, a identificação de seqüências de eventos redundantes exige uma atenção especial. O algoritmo proposto por (HAMOU-LHADJ; LETHBRIDGE, 2002) é capaz de reconhecer seqüências de chamadas que se repetem

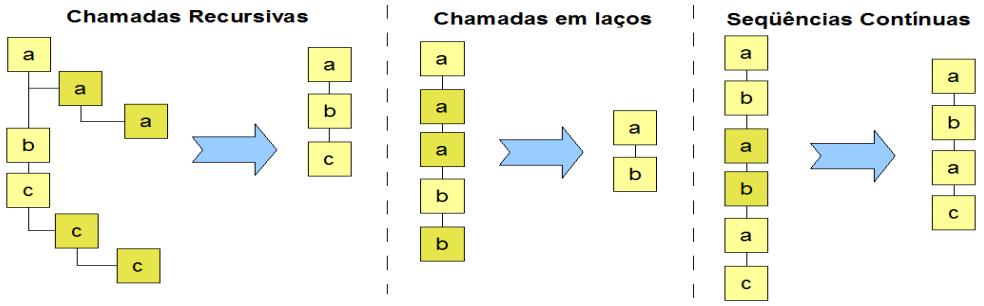


Figura 3.19: Eventos redundantes removidos do rastro com base no algoritmo implementado e adaptado de (HAMOU-LHADJ; LETHBRIDGE, 2002).

continuamente e foi projetado para tratar grande volumes de dados, um dos problemas principais em rastros de execução. A opção adotada na versão corrente foi a de gerar um rastro comprimido compatível com o rastro original. Assim, quando alguma redundância acontece no rastro, ela é removida e apenas um representante é mantido no mesmo. A Figura 3.19 ilustra o efeito da aplicação do algoritmo aos eventos no rastro.

A compressão é um passo opcional para o processo de análise do rastro e visa reduzir a influência de eventos redundantes como os descritos. Isso é justificado pelo fato de que elementos responsáveis pelas redundâncias nem sempre implicam em elementos chaves para a compreensão das características envolvidas. Outra motivação para o uso da compressão está na melhoria no desempenho causada pela redução no tamanho do rastro e conseqüentemente na quantidade de eventos a serem processados pela ferramenta de análise. Porém, maiores estudos ainda precisam ser realizados para confirmar se o rastro original pode ser substituído pelo rastro comprimido sem maiores perdas. Outra alternativa está na extensão da ferramenta de análise para reconhecer os elementos redundantes e dessa forma poder manter no rastro informações sobre quais trechos correspondem a essas ocorrências. Apenas como ilustração a Tabela 3.2 mostra o número de eventos envolvidos no rastro de execução gerado pelo primeiro roteiro definido no análise da ferramenta ArgoUML (que será visto no Capítulo 5) bem como o efeito da compressão obtida sobre o rastro. Como será visto, a Thread-2 que contém eventos responsáveis pela interação principal com o usuário é uma das que obteve a maior taxa de compressão sendo conseqüentemente a maior beneficiada com o processo. Isso mostra que, uma vez comprovado que não há maiores perdas em utilizar o rastro comprimido em lugar do rastro original, a compressão passa a ser um etapa

Tabela 3.2: Compressão obtida sobre um rastro de execução gerado pelo ArgoUML.

	Thread-1	*Thread-2*	*Thread-3*	*Thread-4*	*Thread-5*
Rastro Original	100.255	57.580	204	15.933	7.053
Rastro Comprimido	54.789	11.294	185	5.534	6.634
Taxa de Compressão	45,35%	80,39%	9,31%	65,27%	5,94%

essencial, tanto por permitir lidar com um rastro de menor volume quanto para a obtenção de um melhor desempenho no processamento do rastro pela ferramenta de análise.

5.3 Subsistema de Análise (Trace Analyser)

O subsistema de análise é o elemento central na ferramenta e na abordagem. Nele estão todas as funcionalidades implementadas para lidar com o rastro de execução e extrair informações importantes relacionando o modelo de características e o modelo de código fonte. É dividido em três módulos principais: o módulo de visualização, o módulo de processamento e o módulo de modelos. As seções a seguir discutirão cada um desses módulos.

5.3.1 Visualização

O módulo de visualização é responsável por apresentar e controlar as interfaces de interação com o usuário. Este módulo faz uso tanto de elementos do módulo de processamento, quanto de elementos do módulo de modelos. Seus componentes principais lidam com a visualização do modelo de código fonte, do modelo de características, do rastro de execução e das métricas apresentadas na forma de matrizes.

5.3.1.1 Aplicação

O pacote `traceanalyser.visualization.application` contém o código responsável por coordenar a instanciação e gerenciamento dos demais módulos da aplicação. É o ponto de partida para a execução do analisador.

5.3.1.2 Visualização do Modelo de Código Fonte

O pacote `traceanalyser.visualization.source_code` contém o módulo de visualização do modelo de código fonte. A principal classe no pacote é `SourceCodeModelViewer` que representa o componente de visualização do código fonte.

5.3.1.3 Visualização do Modelo de Características

A visualização do modelo de características é responsabilidade do pacote `traceanalyser.visualization.features`. A classe `FeatureModelViewer` é o elemento principal neste pacote.

5.3.1.4 Visualização do Rastro de Execução

O rastro de execução é visualizado através dos elementos do pacote `traceanalyser.visualization.trace`. A classe `TraceFileViewer` é o elemento chave nesse pacote.

5.3.1.5 Visualização das Métricas (Matrizes)

A visualização das métricas está concentrada especialmente no pacote `traceanalyser.visualization.metrics`. A classe `MetricMatrixPlotter` estende `ColorMatrixPlotter` para suportar a coloração das células das matrizes com base no valor das métricas. A interface `ImageBuildingListener` é utilizada para notificar ouvintes (padrão Observer) sobre a atualização e progresso na geração da matriz. O subpacote `colormaps` contém classes responsáveis pela geração de cores para as células com base em diferentes escalas de cores e que devem estender a classe `AbstractColorFactory`. Finalmente, no subpacote `filters` encontram-se as classes de interface gráfica responsáveis por auxiliar na configuração dos filtros aplicados sobre os elementos da matriz.

5.3.2 Processamento

No módulo de processamento são considerados os diferentes mecanismos para cálculo e extração das métricas.

5.3.2.1 Carregamento

No pacote `traceanalyser.processing.loading` estão os elementos responsáveis por lidar com os arquivos gerados pelo subsistema de instrumentação, incluindo rastros de execução e arquivos de marcação.

5.3.2.2 Indexação do Rastro

Em função do grande volume de dados um esquema de indexação foi utilizado para tornar mais eficiente o acesso e processamento dos elementos presentes no rastro. A indexação permite a geração de matrizes em um tempo razoável para o usuário. Há dois tipos de índice: índice de localização no rastro e índice de mapeamento entre elementos do rastro. No índice de localização, as características, passos de execução e elementos do código fonte são indexados com base nas linhas que ocorrem no rastro de execução. Os índices especificam os intervalos de linhas onde cada elemento ocorre no rastro. Além de permitir a localização mais eficiente dos elementos no rastro de execução, os índices de localização formam a base para a construção dos índices de mapeamento. Estes servem para mapear a relação entre elementos do modelo do código fonte e elementos do modelo de características. A partir da localização indexada, algoritmos de interseção são utilizados para definir o relacionamento entre os elementos.

O pacote `traceanalyser.processing.indexers` contém as classe que fornecem os índices dos elemento envolvidos no rastro. A Figura 3.20 resume as estrutura de classes no pacote.

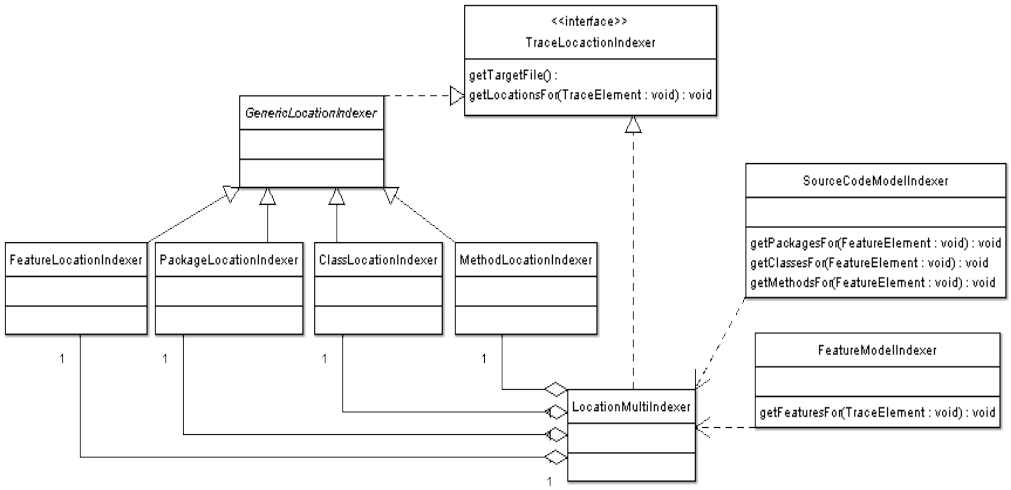


Figura 3.20: Classes de indexação no pacote `traceanalyser.processing.indexers`.

5.3.2.3 Cálculo de Métricas

Como a geração das matrizes demanda um processamento considerável, cada métrica é calculada sob demanda. À medida que o usuário caminha pelos elementos do código e expande a matriz, os elementos envolvidos tem suas métricas calculadas, o que reduz bastante a espera do usuário entre as atualizações da matriz de visualização. O pacote `traceanalyser.processing.metrics` guarda a lógica de cálculo das métricas, bem como a abstração para métrica expressa na classe `Metric` e em suas extensões (Figura 3.21). Cada métrica definida na Tabela 3.1 tem seu próprio extrator os quais estão definidos no

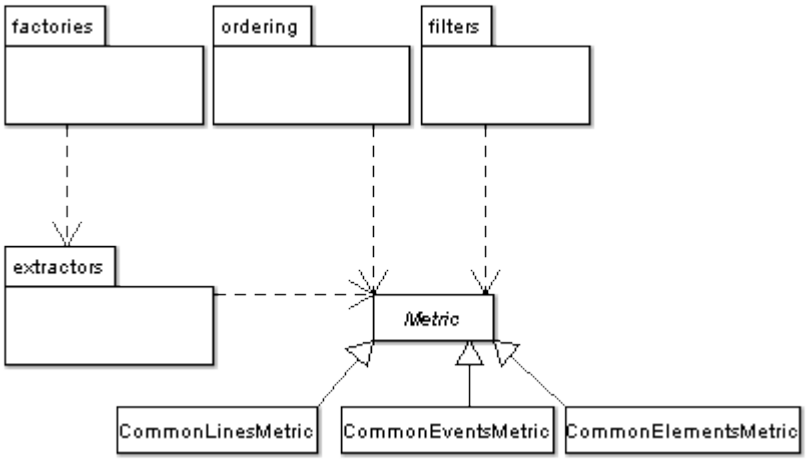


Figura 3.21: Subpacotes e classes do pacote `traceanalyser.processing.metrics`.

subpacote `extractors`. O subpacote `factories` contém as classes responsáveis por instanciar os extratores apropriados para cada tipo de matriz (Cobertura e Similaridade). Os subpacotes `ordering` e `filters` contém abstrações para a lógica de comparação e filtragem de elementos das matrizes de visualização.

5.3.2.4 Estratégia de Cache

Apesar de o cálculo das métricas ser feito sob demanda, a adoção de uma estratégia de cache para tornar mais eficiente a geração das matriz foi necessária. Assim, toda métrica calculada é armazenada em cache para acessos posteriores o que contribui ainda mais para a redução do tempo de espera do usuário. O pacote `traceanalyser.processing.cache` contém as classes responsáveis por manter o cache das métricas. A Figura 3.22 mostra a lógica de implementação do cache baseada no padrão *Proxy (GAMMA et al, 1995)*.

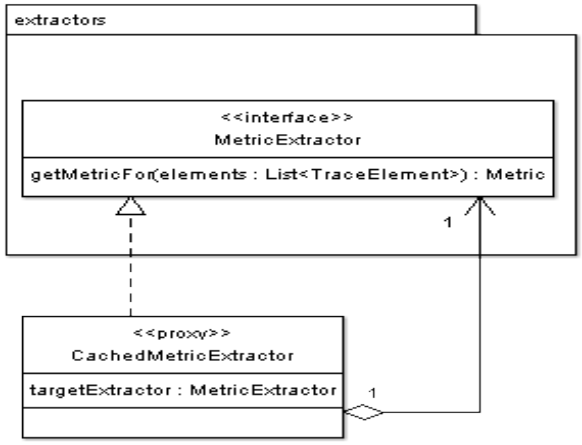


Figura 3.22: Padrão *Proxy* (GAMMA et al, 1995) utilizado como base para a estratégia de Cache.

5.3.3 Modelo

No módulo de modelo são mantidas as abstrações e a lógica para o tratamento de elementos do código fonte, do modelo de características e do rastro de execução.

5.3.3.1 Modelo do Código Fonte

O modelo de código fonte, contido no pacote `traceanalyser.model.source_code`, possui abstrações para os elementos do código atualmente tratados pela ferramenta, tais como, pacotes (`PackageElement`), classes (`ClassElement`) e métodos (`MethodElement`). Todos eles tem como ponto em comum a extensão da classe `TraceElement`, que representa o ponto comum entre os elementos do rastro. A Figura 3.23 resume a relação entre as classes.

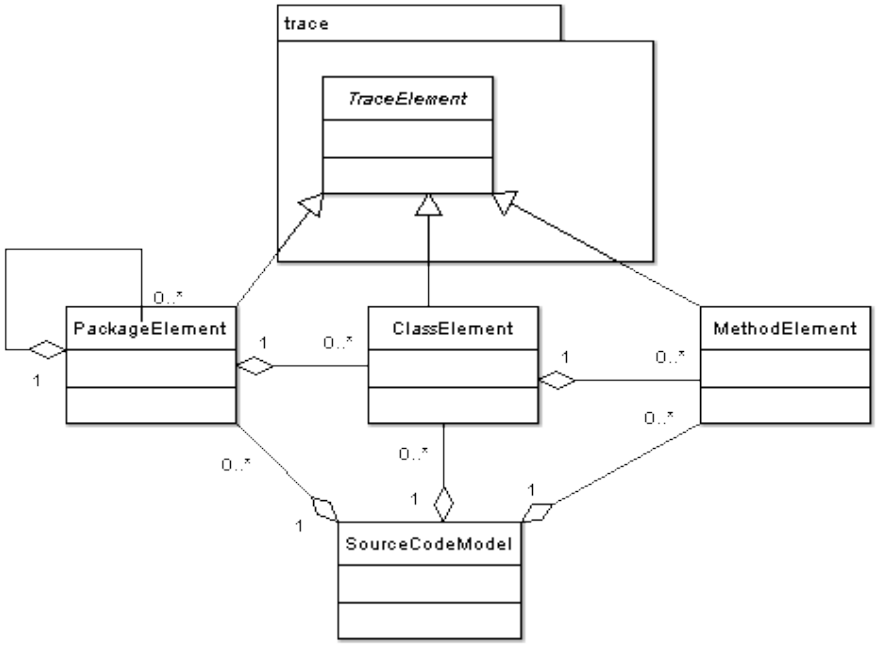


Figura 3.23: Estrutura de classes do pacote `traceanalyser.model.source_code`.

5.3.3.2 Modelo de Características

O modelo de características, contido no pacote `traceanalyser.model.features` (Figura 3.24), reúne as abstrações para características, `FeatureElement`, e para as marcações no rastro, `FeatureExecution`.

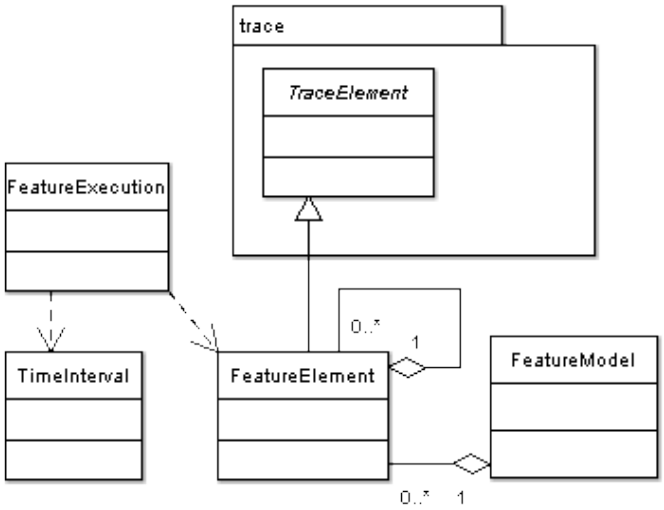


Figura 3.24: Estrutura de classes do pacote `traceanalyser.model.features`.

5.3.3.3 Modelo do Rastro de Execução

O modelo de rastro está localizado no pacote `traceanalyser.model.trace`. Possui a abstração do rastro de execução, `Trace`, a qual é responsável por coordenar as várias operações sobre o rastro, como carregamento, geração de índices, acesso aos elementos do código e características. O subpacote `traceanalyser.model.trace.location` contém as classes responsáveis pela localização dos elementos do rastro, além da lógica para a interseção entre as localizações, sendo esta uma das operações principais na aplicação. A Figura 3.25 resume a estrutura de classes do pacote.

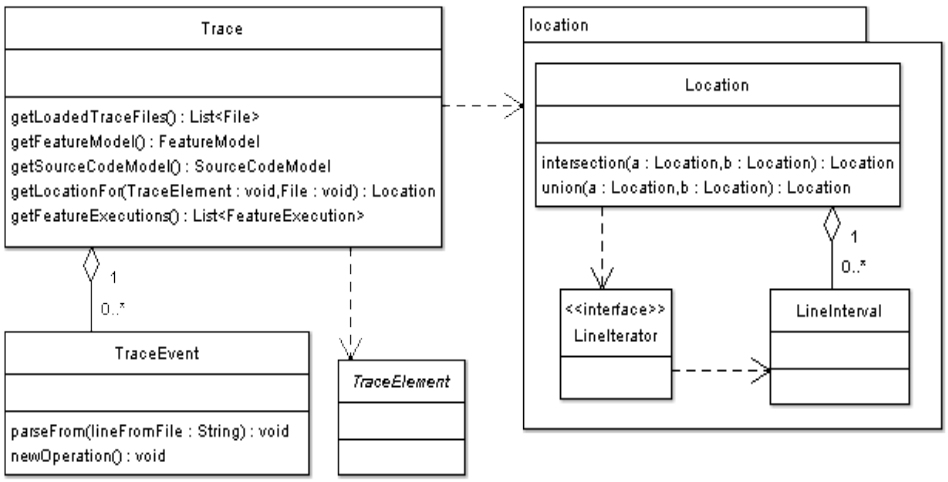


Figura 3.25: Estrutura de classes do pacote `traceanalyser.model.trace`.

6 LIMITAÇÕES E PONTOS DE MELHORIA

Nesta seção algumas das limitações e oportunidades de melhoria na ferramenta serão discutidas. As limitações de cache, persistência de índices, de tratamento de vários rastros, de integração com análise estática, de compressão do rastro e a otimização no cálculo de algumas métricas são apresentadas nas seções 6.1, 6.2, 6.3, 6.4, 6.5 e 6.6, respectivamente.

6.1 CACHE

A estratégia de cache utilizada na implementação corrente cobre apenas o cálculo de métricas. A mesma idéia poderia ser estendida ao cálculo de interseções de forma mais genérica, já que esta é a operação com maior custo no processamento. Outro ponto a ser estudado está na definição dos limites de espaço ideais para o cache e nas estratégias de substituição dos elementos, dado que os elementos são acessados de forma quase aleatória pela expansão dos nós de árvores (modelo de código fonte e características).

6.2 PERSISTÊNCIA DOS ÍNDICES

A implementação atual ainda necessita fazer a persistência dos índices calculados. A cada inicialização da ferramenta os cálculos para o rastro são refeitos. Dado que o rastro não é alterado e o analista pode desejar trabalhar sobre eles durante várias sessões, seria muito útil fazer a persistência dos índices de forma eficiente, para que todos os cálculos sejam feitos uma única vez. Um detalhe que deve ser considerado é a maleabilidade desses índices mediante a alteração do modelo de características utilizado (arquivo *"feature_model.txt"*). Uma estratégia simples seria descartar o índice anterior e recalcular o índice sempre que o arquivo fosse alterado. Porém, considerando que os passos de execução sejam mantidos, bastaria fazer a atualização dos índices para as características alteradas.

6.3 Tratamento de Vários rastros

A implementação corrente lida apenas com um rastro de execução por vez. O tratamento de vários rastros é uma opção bastante útil e que pode trazer grandes ganhos para a análise. Esta opção faz-se necessária especialmente para permitir lidar com vários roteiros de execução que certamente terão vários pontos em comum. Assim, o suporte para o tratamento de vários rastros simultâneos é uma melhoria importante a ser considerada na ferramenta.

6.4 Integração com análise estática

A ferramenta lida apenas com informações extraídas do rastro. Seria importante também integrar essas informações com métricas extraídas por análise estática. A análise de cobertura do código fonte é uma das aplicações diretas para essa funcionalidade.

6.5 Compressão do Rastro

A ferramenta não distingue entre o rastro comprimido e o rastro original. Embora possa funcionar para alguns casos, seria interessante estender a ferramenta para permitir o reconhecimento do rastro comprimido e possivelmente dos pontos onde ocorreu a compressão. Para isso, é possível que as métricas extraídas sejam devidamente adaptadas para levar em conta o efeito da compressão, por exemplo através de um mecanismo de ponderação dos eventos comprimidos, o qual poderia ser habilitado ou desabilitado pelo analista. Outra alternativa está no uso de rastros sumarizados propostos no trabalho de Hamou-LhadJ e Lethbridge (HAMOU-LHADJ; LETHBRIDGE, 2006).

6.6 OTIMIZAÇÃO NO CÁLCULO DE ALGUMAS MÉTRICAS

O tempo de resposta do sistema poderia ser consideravelmente melhorado pela aplicação de otimizações no cálculo das métricas. Essas otimizações poderiam compreender desde a utilização de estruturas de dados mais eficientes, até estratégias de seleção de quais cálculos realmente precisam ser realizados.

7 RESUMO DO CAPÍTULO

Neste capítulo a ferramenta desenvolvida, *Featincode*, foi apresentada. Suas características e funcionalidades foram descritas em detalhes. Exemplos de sua aplicação a um sistema real, *Eye of The Tjger*, foram utilizados. A arquitetura formada pelos três subsistemas (instrumentação, compressão e análise) foi discutida, juntamente com alguns detalhes de implementação. A ênfase maior esteve sobre o subsistema de análise, o qual é o elemento principal da ferramenta. A final do capítulo, algumas limitações e oportunidades de melhorias foram sugeridas.

CAPÍTULO 4. METODOLOGIA DE ANÁLISE

1 OBJETIVOS

Neste capítulo a abordagem proposta para a realização de análises visuais com base no rastro de execução do sistema será apresentada. O processo e os diferentes tipos de análise possíveis serão descritos e detalhados através de exemplos de sua aplicação a um sistema real de pequeno porte.

O objetivo final é entender como um conjunto de características se espalha no código fonte de um sistema, com o intuito de apoiar atividades de evolução, compreensão ou manutenção. Para isso, análises sobre a participação de elementos do código nas características são realizadas, bem como análises de similaridade entre características (em relação aos elementos do código compartilhados) e análises de similaridade entre elementos dos código fonte (em relação às características compartilhadas). Essas análises baseiam-se na relação entre elementos do código fonte e características derivadas do rastro de execução.

Seção 2 mostra algumas das motivações que levaram a criação de *Featincode* e da metodologia de apoio. Seção 3 sintetiza e mapeia algumas das informações principais disponíveis na ferramenta. Seção 4 aborda as etapas para análise e utilização da ferramenta. Seção 5 traz as considerações finais.

2 MOTIVAÇÕES

Durante a realização de atividades de manutenção é fundamental que o desenvolvedor consiga alternar de forma eficiente entre um modelo que expressa o conhecimento do usuário sobre o sistema (modelo de características) e o modelo mental que ele faz do sistema (implementação). Para isso, mecanismos que permitam fazer o mapeamento entre características e implementação são importantes. Idealmente, tal mapeamento deveria ser feito automaticamente e com um custo mínimo para os analistas/desenvolvedores.

Uma vez recuperado o mapeamento entre requisitos e implementação, é possível reconhecer como as características se espalham pelo sistema. Isto pode dar indícios de sistemas mal estruturados, por exemplo, quando uma característica deveria estar compartimentada em unidades coesas do sistema, porém sua implementação se encontra espalhada em diversas unidades. Possíveis complicações na manutenção do sistema podem ser apontadas quando as visões de características e de implementação estão muito distantes devido a um alto grau de espalhamento. Por fim, tomando consciência sobre como as características são implementadas, o desenvolvedor será capaz de coisas como: limitar o escopo da análise e reduzir consideravelmente o esforço para a compreensão, saber as partes afetadas por uma alteração e estimar o impacto de uma mudança no sistema.

A existência de processamento concorrente na execução é outro ponto importante sobre a metodologia proposta. Tendo consciência do comportamento concorrente do sistema e sabendo o significado e a função das várias *threads*, o desenvolvedor ganha novas perspectivas em atividades de manutenção (correção, refatoração, evolução).

O custo e o esforço para a compreensão do sistema pode potencialmente ser diminuído uma vez que *threads* não interessantes para a análise sejam identificadas e, conseqüentemente, os elementos associados a essas *threads* sejam desconsiderados da análise do código fonte e do rastro de execução.

3 Informações x Fontes

Capítulo 3 mostra as características da ferramenta *Featincode* e com muitas informações podem ser extraídas da ferramenta, algumas disponíveis em mais de um local. Para apoiar as análises, a Tabela 4.1 sintetiza as informações e as fontes de onde podem ser obtidas. A tabela é complementada com um breve resumo sobre cada informação nos parágrafos a seguir.

Características / Eventos Rastro: trata do grau de participação da característica no rastro. Esta informação está disponível nos gráficos de barras de cada *thread* apresentados no *TraceFileViewer*. Nestes gráficos são mostrados o percentual de ocorrência das características ou passos de execução em intervalos de eventos separadas por *thread*.

Tabela 4.1: Informações x Fonte na ferramenta *Featincode*.

	Gráfico Barras Threads	*Vis. Modelo Elem. Código*	*Vis. Modelo Característica*	*Matriz Cobertura Característica*	*Matriz Cobertura Elem. Código*	*Matriz Similaridade Característica*	*Matriz Similaridade Elem. Código*
Características / Eventos Rastro	x	..	..	..	..	..	..
Elem. Código / Eventos Rastro	x	..	..	..	..	..	..
Características x Elem. Código	x	..	..	x	x	..	..
Similaridade Características	x	..	..	..	..	x	..
Similaridade Elem. Código	x	..	..	..	..	..	x
Tamanho Threads	x	..	..	..	..	..	..
Métricas Rastro Características	..	x	..	x	..	x	..
Métricas Rastro Elem. Código	..	..	x	..	x	..	x
Características Exclusivas	..	..	x	..	..	..	..
Elem. Código Exclusivos	..	x	..	..	..	..	..

Elem. Código / Eventos Rastro: os eventos no rastro associados a cada elemento do código (pacote, classe ou método) podem ser visualizados nos gráficos de barras do *TraceFileViewer* no mesmo formato utilizado para as características (descrito no item anterior).

Características x Elem. Código: o cruzamento entre eventos associados às características e eventos associados aos elementos do código pode ser visualizado em três lugares. O *TraceFileViewer* exibe gráficos de barras e valores percentuais sobre o número de eventos compartilhados no rastro por características e elementos do código. As matrizes exibindo métricas de cobertura de características mostram o grau de participação dos elementos do código na execução das características quanto ao número de eventos (NSEPF, NSECF e NSEMF), pacotes (NPPF), classes (NPCF) e métodos (NPMF). Por sua vez, as matrizes exibindo métricas de cobertura para elementos do código exibem a relação inversa, mostrando o grau de espalhamento das características pelos elementos do código fonte em termos de eventos (NSEFP, NSEFC e NSEFM), pacotes (NSPFP), classes (NSMFP) e métodos (NSMFP e NSMFC). O cruzamento pode ser feito no escopo de uma ou mais *threads*.

Similaridade Características: a similaridade entre características pode ser visualizada através das matrizes com métricas de similaridade entre características. Os critérios de comparação podem ser o número de pacotes (NSPF), classes

(NSCF) ou métodos (NSMF) compartilhados entre as características. Outra opção, consiste na visualização dos eventos compartilhados entre características através dos gráficos de barras no *TraceFileViewer*.

Similaridade Elem. Código: por sua vez, a similaridade entre elementos do código pode ser exibida através das matrizes com métricas de similaridade entre elementos do código. Neste caso, os critérios são o número de características compartilhadas entre pacotes (NSFP), classes (NSFC) ou métodos (NSFM). Como para características, os eventos compartilhados entre elementos código fonte também podem ser visualizados do *TraceFileViewer*.

Tamanho Threads: a quantidade de *threads* iniciadas durante a execução bem como o número de eventos gerados por cada uma delas pode ser observado no *TraceFileViewer*. Cada painel representa uma *thread* iniciada e o limite superior do último intervalo de eventos representa a quantidade de eventos gerados pela respectiva *thread*.

Métricas Rastro Características: o número de passos de execução envolvidos numa característica pode ser visualizado através do *FeatureModelViewer* pela seleção de uma das características no modelo. Além disso, há a opção detalhamento das métricas exibidas sobre uma característica pela seleção de células nas matrizes de visualização.

Métricas Rastro Elem. Código: métricas sobre os rastro para os elementos do código também estão disponíveis através do *SourceModelViewer*. Por exemplo, para pacotes é possível ver o número de subpacotes não vazios (contendo classes ou interfaces diretas), número total de classes diretas e/ou indiretas, além do número de métodos associados (devido a classes/interfaces diretas ou indiretas). As matrizes também são importantes fontes de métricas sobre elementos do código, as quais são visualizadas através da seleção de células e detalhamento das métricas.

Características Exclusivas: características associadas exclusivamente a um dado elemento do código podem ser listadas através de menus de opções disponíveis após a seleção do elemento no *SourceCodeModel*. É possível definir como escopo uma ou mais *threads*.

Elem. Código Exclusivos: de forma semelhante ao item anterior, há a opção de listar elementos do código (pacotes, classes ou métodos) exclusivos de uma característica (ou passo de execução) selecionada no FeatureModelViewer. O

escopo também pode ser definido para uma ou mais *threads*.

4 Processo de Análise

As etapas envolvidas no processo de análise consistem em:

1. Definição dos Objetivos da Análise
2. Definição do Modelo de Características de Interesse
3. Planejamento dos Roteiros de Execução
4. Instrumentação do Sistema
5. Coleta do Rastro
6. Análises Visuais

Para ilustrar a abordagem, um exemplo simples será utilizado sobre *Eye of the Tjger*, um jogo criado com o *framework Tjger, **T**he **J**ava **G**ame **E**ngine to **R**euse* (GRUBER; JORDAN, 2008), que é uma aplicação de demonstração das funcionalidades deste *framework*. A aplicação consiste em um jogo de cartas com dados. O número de participantes pode variar de 2 a 8 jogadores. As regras do jogo são simples: cada jogador recebe três cartas no início de cada rodada. A jogada consiste no lançamento de um dado e na escolha de uma das cartas. O valor da carta (2 a 15) é então multiplicado pelo valor do dado. A rodada termina quando todas as cartas são jogadas. O jogador com maior pontuação ao final de todas as rodadas é o vencedor. As seções a seguir detalharão as etapas citadas e as análises visuais propostas.

As seções a seguir detalharão as etapas citadas (seções 4.1, 4.2, 4.3, 4.4 e 4.5, respectivamente) e as análises visuais propostas (Seção 4.6).

4.1 Definição dos Objetivos da Análise

O estabelecimento dos objetivos da análise consiste no primeiro requisito para a utilização da abordagem. Esses objetivos serão o ponto de partida para as etapas posteriores e definirão o escopo da análise. A abordagem pode ser empregada para apoiar atividades de compreensão e manutenção.

Quando o objetivo da análise consiste em auxiliar a compreensão, as características do sistema a serem compreendidas e o grau de entendimento

almejado devem ser definidos. Para um entendimento de alto nível, é possível que apenas a análise de pacotes possa ser suficiente. Por outro lado, para um entendimento aprofundado, as classes e métodos envolvidos nas características deverão ser incluídos na análise.

Quando a intenção é apoiar atividades de manutenção, é preciso definir que tipo de manutenção deverá ser efetuada (corretiva, evolutiva, perfectiva). Para a manutenção corretiva, o escopo da análise provavelmente será limitado às características envolvidas em uma falha ou erro na implementação do sistema. Uma vez identificados os elementos envolvidos nessas características, o analista terá um bom ponto de partida para proceder com a correção. Na manutenção evolutiva, a abordagem poderá auxiliar na identificação de elementos chaves de características similares ou características relacionadas àquelas que devem ser inseridas no sistema. Por fim, para a manutenção perfectiva, a abordagem pode fornecer informações relativas à coesão entre elementos do código em termos de características implementadas, por exemplo, pela observação do espalhamento das características pelos elementos do código.

No caso da aplicação *Eye of the Tjger*, supõe-se que os objetivos do desenvolvedor sejam entender como o *framework Tjger* é utilizado pela aplicação e onde as características da aplicação são implementadas no código.

4.2 Definição do Modelo de Características de Interesse

O segundo requisito para a utilização da abordagem consiste na definição de um modelo de características do sistema. Este modelo servirá de guia para o desenvolvedor durante o planejamento e a coleta do rastro de execução. Além disso, esse modelo de características será também utilizado durante as análises.

Para o estabelecimento desse modelo, técnicas tradicionais provindas da área de Engenharia de Domínio e Modelagem de Características podem ser utilizadas (KANG et al, 1990)(CZARNECKI; EISENNECKER, 2000). Nesse sentido, alguns recursos e fontes de informação deverão ser buscados, tais como: a documentação do sistema (de projeto e de usuário), histórico de versões, usuários do sistema, modelos de domínio, sistemas similares, dentre outros.

O escopo do modelo pode consistir em um subconjunto específico de características ou do conjunto total de características do sistema. A abrangência do

escopo dependerá dos objetivos da análise e das atividades que essa análise pretende apoiar (compreensão ou manutenção). É importante destacar que esse modelo pode ser reutilizado em outras sessões de análise ou mesmo em outras versões do sistema. Assim, é possível que essa etapa já tenha sido realizada anteriormente.

Quanto à seleção das características de interesse, o ideal é que o analista enfoque naquelas mais alinhadas aos seus objetivos.

A Figura 4.1 mostra parte do modelo de características para a aplicação *Eye of the Tjger* definido após a análise da documentação de usuário, pela utilização da aplicação e com base nos objetivos estabelecidos na seção anterior.

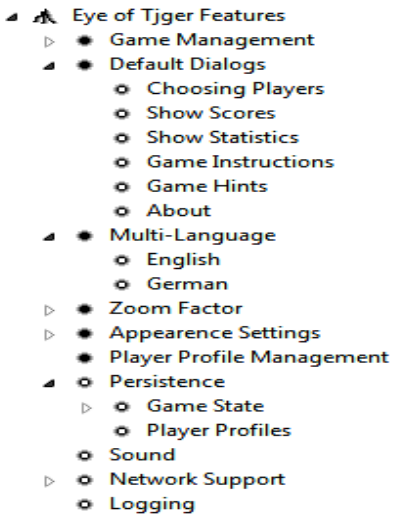


Figura 4.1: Parte do Modelo de Características para a aplicação Eye of the Tjger.

4.3 Planejamento dos Roteiros de Execução

De posse do modelo de características, o analista deve então definir quais roteiros de execução são necessários para exercitar as características de interesse que estão nas folhas da árvore representando o modelo de características. Para isso a consulta à documentação e aos usuários do sistema pode ser necessária. O roteiro de execução deverá conter todos os passos necessários para o exercício das características de interesse. A Figura 4.2 ilustra a relação entre algumas características da aplicação *Eye of The Tjger* e os passos para exercitar essas características relativos a um dos roteiros de execução definidos a seguir.

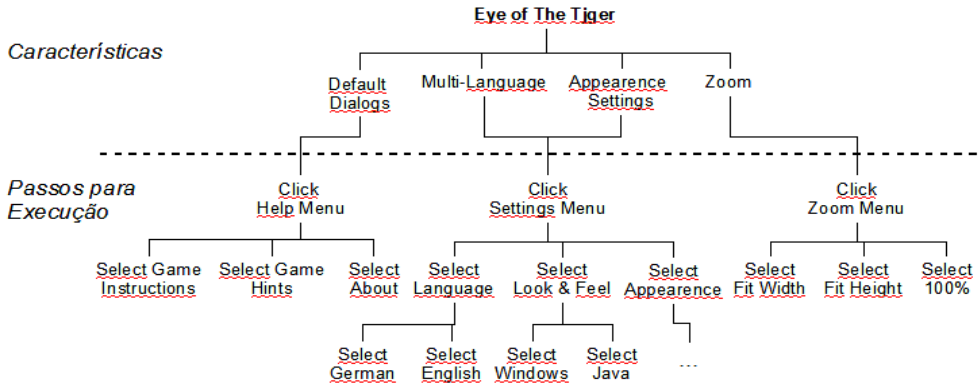


Figura 4.2: Relacionamento entre características e passos de execução para o *Roteiro de Execução 1*.

O Roteiro 1 mostrado na Tabela 4.2 foi definido com o objetivo de exercitar algumas das características de configuração disponíveis na aplicação *Eye of the Tjger* derivadas do framework *Tjger*. A característica exercitada em cada passo do roteiro está marcada na tabela. Após a inicialização da aplicação (inserida no roteiro apenas para fins ilustrativos e que não tem necessariamente alguma relação com as características exercitadas), alguns diálogos e opções de configuração são acessados partindo da seleção das opções do menu. Alguns termos envolvidos nos nomes dos passos de execução mostrados na Figura 4.2 foram simplificados na Tabela 4.2 pelas limitações de espaço.

Tabela 4.2: *Roteiro de Execução 1* envolvendo algumas características de configuração do *Eye of the Tjger*.

		Default Dialogs	Language	Appearence Settings	Zoom
1	*Initialization*				
2	*Help Menu -> Select Game Instructions*	x			
3	*Help Menu -> Select Game Hints*	x			
4	*Help Menu -> Select About*	x			
5	*Settings Menu -> Language -> Select German*		x		
6	*Settings Menu -> Language -> Select English*		x		
7	*Settings Menu -> Look & Feel -> Select Windows*			x	
8	*Settings Menu -> Look & Feel -> Select Java*			x	
9	*Settings Menu -> Appearence -> Background -> Select Carpet*			x	
10	*Settings Menu -> Appearence -> Background -> Select Green*			x	
11	*Settings Menu -> Appearence -> Game Board -> Select Glass*			x	
12	*Settings Menu -> Appearence -> Game Board -> Select Wood*			x	
13	*Settings Menu -> Appearence -> Table Decoration -> Select Flowers*			x	
14	*Settings -> Appearence -> Table Decoration -> Select Candle*			x	
15	*Zoom Menu -> Select Fit Width*				x
16	*Zoom Menu -> Select 100%*				x
17	*Zoom Menu -> Select Fit Height*				x

O *Roteiro 2* mostrado na Tabela 4.3 representa uma partida do jogo. No passo

"Game Menu -> New Game -> Game Configuration" um novo jogo é criado e configurado. O jogo é configurado para um usuário enfrentar o computador, consistindo em um única rodada. Em *"Settings Menu -> Game -> Disable Sound"* o som emitido após certas ações no jogo é desabilitado. Os passos *"P1 -> Click Roll dice 1"* e *"P1 -> Select Card (1)"* representam ao início da jogada do usuário (*"P1" = Player 1*) e correspondem ao lançamento do dado e seleção da carta, respectivamente. O passo *"P1 -> Click Play Card (1)"* envolve a finalização da jogada do usuário com a confirmação da carta escolhida. Este passo é seguido automaticamente pela jogada do computador. Em *"Settings Menu -> Game -> Enable Sound"* o som é habilitado e a partir daí algumas ações como o lançamento do dado e o processamento da jogada são acompanhados de um efeito sonoro. O usuário realiza uma nova jogada seguida pela jogada do computador a partir de *"P1 -> Click Roll dice (2)"*. O usuário e o computador realizam sua última jogada que é seguida pela finalização da rodada/partida. Por fim, os diálogos de informação sobre a partida, estatísticas e resultados são acessados a partir do passo *"Game Menu -> Game Information -> Current Game"*.

Tabela 4.3: *Roteiro de Execução 2* com a seqüência de uma partida de *Eye of The Tjger*.

		Sound	Game Play	Default Dialogs
1	*Initialization*			
2	*Settings Menu -> Game -> Disable Sound*	x		
3	*Game Menu -> New Game -> Game Configuration*		x	
4	*P1 -> Click Roll dice (1)*		x	
5	*P1 -> Select Card (1)*		x	
6	*P1 -> Click Play Card (1)*		x	
7	*Settings Menu -> Game -> Enable Sound*	x		
8	*P1 -> Click Roll dice (2)*		x	
9	*P1 -> Select Card (2)*		x	
10	*P1 -> Click Play Card (2)*		x	
11	*P1 -> Click Roll dice (3)*		x	
12	*P1 -> Select Card (3)*		x	
13	*P1 -> Click Play Card (3)*		x	
14	*Game Menu -> Game Information -> Current Game*			x
15	*Game Menu -> Game Information -> Statistics*			x
16	*Game Menu -> Game Information -> High Score*			x

4.4 INSTRUMENTAÇÃO DO SISTEMA

O subsistema de *Featincode*, *TraceExtractor* é responsável pela coleta de rastros e instrumentação do sistema alvo da análise, tendo sido desenvolvido com

base em *AspectJ*. Em princípio, o código fonte do sistema ou seus binários (.class ou .jar) devem estar disponíveis, assim como o código de bibliotecas utilizadas pelo sistema cuja interação deva ser analisada. O sistema alvo deve ser recompilado com suporte a aspectos para a coleta do rastro pela ferramenta de instrumentação. Nenhuma alteração adicional é necessária no código fonte, exceto pela alteração do projeto no ambiente de desenvolvimento para dar suporte a *AspectJ* e para vincular o aspecto de instrumentação ao sistema alvo.

A instrumentação da aplicação Eye of the Tjger foi feita em quatro passos:

1. importando seu código fonte para o Eclipse 3.3;
2. alterando a natureza do projeto criado para projeto baseado em AspectJ;
3. adicionando a ferramenta de instrumentação (*tracer.jar*) ao *AspectJ In Path*;
4. considerando o interesse na interação com o *framework* base, a biblioteca correspondente (*tjger.jar*) também foi adicionada ao *AspectJ In Path*.

4.5 Coleta do Rastro

Com base nos roteiros de execução definidos, o sistema alvo instrumentado deve ser executado. Durante a execução, o analista deve fazer a marcação dos instantes de início e fim dos passos de execução relativos aos roteiros de execução vinculados às características *"folhas"*. As características *"internas"* da árvore de característica são vinculadas aos passos de suas sub-características. Com base nesta marcação, os eventos (chamadas de métodos) ocorridos são relacionados às características correspondentes.

Nesta etapa, a aplicação *Eye of the Tjger* foi executada. Para cada roteiro definido nas Tabela 4.2 e 4.3 anteriores, uma execução da aplicação foi realizada. As Tabelas 4.4, 4.6 e 4.5 exibem algumas estatísticas extraídas do rastro com o auxílio da ferramenta de análise desenvolvida, apresentando o número de eventos gerados, *threads* iniciadas e elementos do código fonte envolvidos nas execuções.

Tabela 4.4: Cobertura do código pelos roteiros de execução.

	Pacotes	Classes	Métodos	Eventos	Threads
Cenário 1	19	100	851	737.700	8
Cenário 2	19	150	1.201	862.570	11

Tabela 4.5: Estatísticas de tempo de execução para o Roteiro 2 (Seqüência da Partida).

Cenário 2 (862.570 eventos)	Pacotes	Classes	Métodos	Eventos
Initialization	19	112	650	44.392
Settings Menu -> Game -> Disable Sound	13	28	98	26.488
Game Menu -> New Game -> Game Configuration	18	89	553	60.388
P1 -> Click Roll dice (1)	14	40	171	18.268
P1 -> Select Card (1)	14	37	178	24.590
P1 -> Click Play Card (1)	14	53	343	54.263
Settings Menu -> Game -> Enable Sound	15	38	165	46.539
P1 -> Click Roll dice (2)	14	44	179	30.718
P1 -> Select Card (2)	14	39	170	23.765
P1 -> Click Play Card (2)	14	56	348	43.283
P1 -> Click Roll dice (3)	14	47	173	61.823
P1 -> Select Card (3)	14	39	161	18.756
P1 -> Click Play Card (3)	14	65	376	64.437
Game Menu -> Game Information -> Current Game	15	59	276	55.219
Game Menu -> Game Information -> Statistics	15	59	276	50.735
Game Menu -> Game Information -> High Score	15	69	276	42.111

Tabela 4.6: Estatísticas de tempo de execução para o Roteiro 1 (Opções de Configuração).

Cenário 1 (737700 eventos)	Pacotes	Classes	Métodos	Eventos
Initialization	19	112	650	36.318
Help Menu -> Game Instructions	14	33	105	21.979
Help Menu -> Game Hints	15	41	132	20.031
Help Menu -> About	12	30	95	21.845
Settings Menu -> Language -> German	14	39	140	24.583
Settings Menu -> Language -> English	14	39	140	25.200
Settings Menu -> Look & Feel -> Windows	13	39	131	30.735
Settings Menu -> Look & Feel -> Java	13	39	131	33.972
Settings Menu -> Appearence -> Background -> Carpet	14	48	275	57.887
Settings Menu -> Appearence -> Background -> Green	14	49	278	43.720
Settings Menu -> Appearence -> Game Board -> Wood	14	48	274	62.974
Settings Menu -> Appearence -> Game Board -> Glass	14	48	275	66.751
Settings Menu -> Appearence -> Table Decoration -> Flowers	14	48	274	63.637
Settings Menu -> Appearence -> Table Decoration -> Candle	14	48	275	88.973
Zoom Menu -> Fit Width	13	30	118	28.855
Zoom Menu -> 100%	13	31	123	17.387
Zoom Menu -> Fit Height	13	30	119	23.309

4.6 ANÁLISES VISUAIS

Os tipos de análises visuais possíveis na abordagem proposta são:

a) Análise do Espalhamento das Características pelo Rastro
b) Análise do Significado e Seleção de *Threads* de Interesse
c) Análise de Elementos do Código relacionados às Características
d) Análise de Similaridade entre Elementos do Código
e) Análise de Similaridade de Implementação entre Características

As cinco formas de análise propostas para a abordagem serão descritas a seguir entre as seções 4.6.1 e 4.6.5.

4.6.1 Análise do Espalhamento das Características pelo Rastro

Durante a execução de um sistema, muitos eventos são gerados, porém nem todos tem relação com as características de interesse. Assim, essa análise tem a finalidade de identificar quais e quantos eventos no rastro tem relação com as características. O universo para análise do espalhamento é formado pelos elementos do código fonte participantes na execução e as características de interesse exercitadas através dos passos de execução.

Após carregar os rastros coletados na ferramenta de análise, os primeiros pontos a serem observados são os elementos do código fonte manifestados no rastro e os passos de execução realizados.

A Figura 4.3 mostra os principais pacotes manifestados no rastro de execução para o Roteiro 1 (Opções de Configuração). Pela seleção dos elementos do código nessa visualização é possível obter estatísticas relativas à execução do referido elemento. Pode ser observado que, para o pacote `tjger`, um total de 10 pacotes (contendo classes ou interfaces manifestadas no rastro), 66 classes+interfaces e 530 métodos estão envolvidos neste rastro de execução. Esses dados fornecem uma primeira referência para o espalhamento das características. Quanto mais elementos presentes (pacotes, classes, métodos) maior é o indício de que há um alto grau de espalhamento dessas características no sistema. Porém, isso

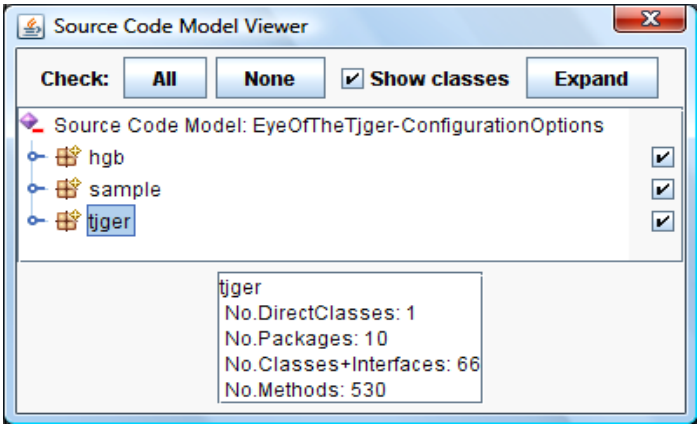


Figura 4.3: Modelo de código fonte extraído do rastro de execução do Roteiro 1 apresentando estatísticas de tempo de execução para o pacote `tjger`.

dependerá fundamentalmente do modelo de características definido, das características exercitadas, do tamanho e complexidade do sistema.

A Figura 4.4 mostra os passos de execução realizados para a geração do rastro.

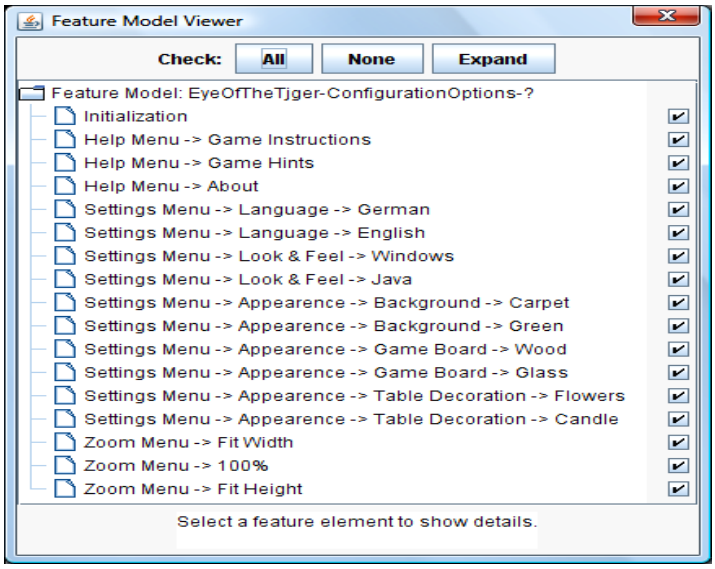


Figura 4.4: Passos de execução referentes ao Roteiro 1.

Alternativamente, estes passos podem ser agrupados e carregados na ferramenta para refletir a hierarquia do modelo de características exercitadas como mostrado na Figura 4.5. Nesse caso, após a seleção de uma das características, a quantidade de passos de execução envolvidos é mostrada. Por exemplo, a quantidade de passos de execução (3) relacionados à característica *"Zoom"* é apresentada após o rótulo *Nro.LeafFeatures*.

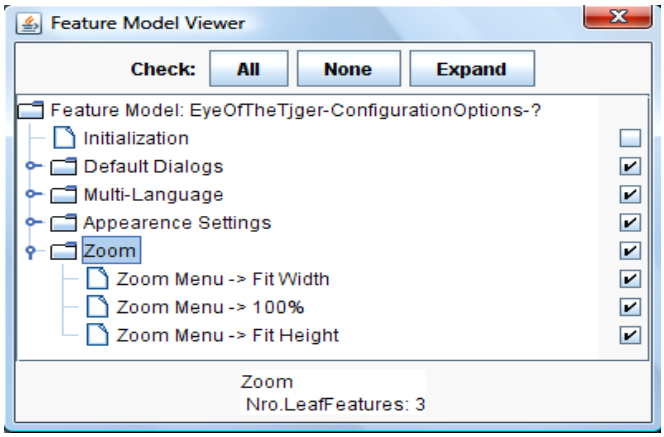


Figura 4.5: Passos de execução referentes ao Roteiro 1, agrupados segundo o modelo de características.

Uma maneira direta de observar o espalhamento de características pelo rastro é através de sua seleção e posterior observação nos gráficos de barras de cada *thread* de execução. A Figura 4.6 mostra o espalhamento de dois passos de execução nas respectivas *threads*. Os passos são a *inicialização* (*"Initialization"*) e o *ajuste da aparência da aplicação para o estilo Windows* (*"Settings Menu -> Look & Feel -> Windows"*), ambos relativos ao Roteiro 1. É possível ver em quais intervalos de eventos essas características ocorrem. Na figura podemos ver, por exemplo, que na *Thread-2*, o *ajuste da aparência para o estilo Windows* ocorre parcialmente entre os eventos de número 150.00 e 190.000 no rastro de execução. É possível observar ainda que esses eventos relativos ao passo de execução correspondem a 3,58% do total de eventos gerados por essa *thread*. Como esperado, não há interseção entre os eventos relacionados à inicialização e os eventos do ajuste de estilo, dado que a marcação desses passos é feita em intervalos disjuntos de tempo.

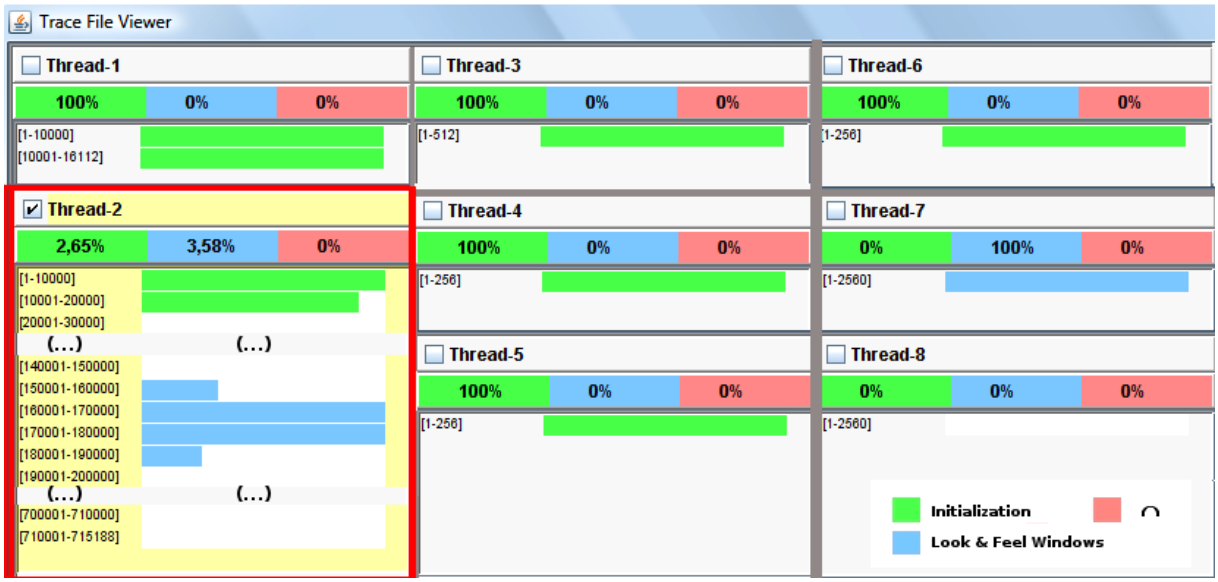


Figura 4.6: Espalhamento da *inicialização* e *ajuste da aparência da aplicação para o estilo Windows* referente à execução do Roteiro 1 (*Opções de Configuração*) da aplicação *Eye of the Tjger*.

4.6.2 Análise do Significado e Seleção de Threads de Interesse

A identificação de quais *threads* foram executadas, qual o papel e a importância de cada uma na execução de um sistema são pontos importantes para a compreensão do mesmo. Nesse sentido, o mapeamento de elementos do código e de características para os eventos gerados por cada *thread* iniciada fornece

informação importante no processo. Além disso, há a possibilidade de se excluir da análise, as *threads* sem eventos relacionados às características, possibilitando assim uma redução no escopo no total de elementos a serem avaliados.

A análise do espalhamento das características feita anteriormente pode fornecer alguma informação sobre a semântica das *threads*. Para o Roteiro 1 e com base na Figura 4.6, a *inicialização* é responsável por 100% dos eventos gerados nas *threads* 1, 3, 4, 5 e 6. Além disso, parte dos eventos no intervalo de 1 a 20.000 da *Thread-2* também fazem parte da *inicialização*. Este é um fato importante para a filtragem de eventos e *threads* de interesse. Caso não exista interesse na análise dos eventos da *inicialização*, é possível descartar a análise de 5 das 8 *threads* iniciadas. Por sua vez, o passo de execução *"Settings Menu -> Look & Feel -> Windows"* é responsável por 100% dos eventos na *Thread-7*. Através da geração de gráficos similares ao da Figura 4.6, foi possível constatar que *"Settings Menu -> Look & Feel -> Java"* é responsável por 100% dos eventos na *Thread-8*. Dessa forma, exceto pelos dois passos de ajuste de estilo (*"Settings Menu -> Look & Feel -> Windows"* e *"Settings Menu -> Look & Feel -> Java"*), a única *thread* de interesse para a análise das demais características exercitadas nesse roteiro é a *Thread-2*. Realizando outras execuções com o mesmo roteiro de execução e com pequenas variações nos passos de execução, observa-se certa constância no número de eventos gerados por cada *thread*. A *Thread-1*, por exemplo, produz praticamente o mesmo número de eventos em todas as execuções. Isso permite inferir que esta *thread* seja responsável pela inicialização em todas as execuções. Assim, a quantidade de eventos associados a uma característica (ou passo de execução) e sua representatividade diante do total de eventos gerados por uma *thread* pode servir como parâmetro para a identificação da semântica dessa *thread* em relação à característica.

Além de identificar a participação das características nos eventos de cada *thread*, um segundo parâmetro, o tamanho da *thread*, pode ser utilizado como base para a escolha das *threads* de interesse. Em geral e segundo observações nos testes realizados, existem *threads* secundárias iniciadas que são ativadas para funções bastante específicas e, portanto, não envolvem uma grande quantidade de eventos ou mesmo um grande número de elementos do código fonte. As *threads* principais da aplicação geralmente contêm o maior número de eventos e estão associadas ao maior número de características exercitadas. Quando os eventos de

uma *thread* estão em pequena quantidade e envolvendo poucos elementos do código fonte, estas podem ser avaliadas rapidamente, muitas vezes observando o rastro bruto para reconhecer o impacto de seus eventos no roteiro de execução. No Roteiro 1, desconsiderando as *threads* associadas exclusivamente à *inicialização* (1 e 3 a 6), restam as *threads* 2, 7 e 8. Como observado anteriormente, as *threads* 7 e 8 estão associadas aos passos de alteração do estilo da aplicação. Examinando os arquivos com os rastros gerados por essas *threads*, observa-se que o único método presente nos eventos é o `filterRGB` da classe `HGBaseBrightImgFilter`.. Assim, constata-se que embora várias *threads* tenham sido iniciadas durante a execução do roteiro, a aplicação não exibe um comportamento essencialmente concorrente, dado que as demais *threads* realizam apenas tarefas complementares à *thread* principal da aplicação (*Thread-2*).

4.6.3 Análise de Elementos do Código relacionados às Características

É esperado que o principal objetivo no uso de *Featincode* esteja em identificar o espalhamento das características de interesse pelo sistema. Para isso, a ferramenta quantifica as relações entre elementos do código fonte ativados durante a execução e os passos de execução para exercitar as características. Os gráficos e matrizes gerados pela ferramenta são os principais recursos para atingir este objetivo.

A análise do visualizador de rastro e das matrizes de visualização pode dar bons indícios do espalhamento das características pelo sistema. Para isso, a combinação de elementos do código fonte e características deve ser o alvo da análise nessas formas de visualização.

Uma primeira medida do espalhamento das características pelo código fonte está na visualização dos elementos do código fonte manifestados no rastro de execução para cada característica. A Figura 4.7 mostra a participação dos pacotes principais (pacotes de mais alto-nível na árvore de elementos de código) manifestados na execução (`hgb`, `sample` e `tjger`) das características para a *mudança da língua da aplicação* (*"Settings Menu->Language-> German"* e *"Settings Menu ->Language ->English"*). A interseção entre os eventos associados aos pacotes e os eventos das características é mostrada em vermelho. Pode ser

Figura 4.7: Visualização dos eventos no rastro associados aos pacotes principais (hgb, sample, tjger) (verde) e as características de mudança de língua (azul) da aplicação Eye of the Tjger na execução do Roteiro 1.

observado que os pacotes `sample` e `tjger` respondem pela maior parte dos eventos no rastro da *Thread-2* (barras em verde) correspondendo a 43,02% e 42,47% do total de eventos, respectivamente. Por sua vez, *"Settings Menu->Language-> German"* e *"Settings Menu ->Language ->English"* são resposáveis por 3,44% e 3,52% dos eventos nessa *thread* (barras em azul). Finalmente, as barras em vermelho indicam o percentual de eventos em comum entre as características e os pacotes. Para as duas características é possível observar que o pacote `sample` responde pela maior parte do eventos, sendo a interseção com *"Settings Menu->Language-> German"* e *"Settings Menu ->Language ->English"* correspondente a 1,7% e 1,77% do total de eventos da *thread,* respectivamente.

As matrizes com métricas de cobertura de características permitem visualizar a contribuição de elementos do código fonte na execução de uma característica. Essa contribuição pode ser observada em diferentes níveis de granularidade,

partindo de eventos (NSEPF, NSECF, NSEMF) e passando por métodos (NPMF), classes (NPCF) e pacotes (NPPF). Por exemplo, a métrica NPCF para um dado pacote permite visualizar sua representatividade em relação ao total de classes participantes na execução de uma característica ou passo de execução. A Figura 4.8 mostra essas diferentes relações entre pacotes e características. As quatro matrizes são estruturalmente equivalentes, dado que todas as métricas são derivadas dos eventos presentes no rastro. O que varia é a intensidade da relação.

Considerando a primeira matriz (Eventos) na Figura 4.8, os pacotes `sample` e `tjger` continuam com maior destaque (células em vermelho), confirmando as observações anteriores (Figura 4.7). A relação geral dos pacotes frente as demais características pode ser vista nas matrizes. Isso permite identificar os sub-pacotes com maior contribuição. Neste caso, os pacotes `sample.gui` e `tjger.gui` são os que mais contribuem em número de eventos depois de `sample` e `tjger`. Para as demais matrizes (Métodos, Classes e Pacotes), a quantidade de ocorrências de um evento no rastro não influencia a métrica, justificando a diferença na intensidade de coloração das células em relação à matriz de Eventos. Nessas matrizes a variedade de elementos distintos do código é levada em conta. Por exemplo, de acordo com a segunda matriz, as classes do pacote `tjger` possuem um número maior de

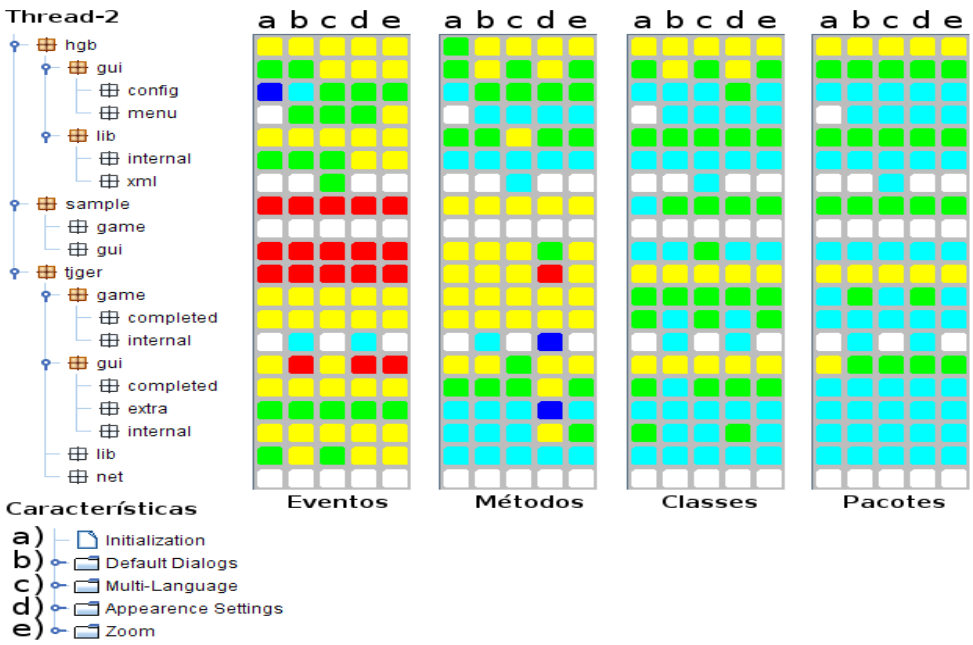


Figura 4.8: Matrizes com métricas de cobertura de características mostrando a representatividade dos pacotes sobre as características no rastro da *Thread-2* do Roteiro 1 em termos de eventos (NSEPF), métodos (NPMF), classes (NPCF) e pacotes (NPPF), respectivamente.

métodos distintos participantes no rastro, em especial para a característica *"Appearence Settings"* (coluna *d*). Para classes, os pacotes `hgb` e `tjger` mostram maior destaque (em amarelo), podendo ainda ser observado que `tjger.gui` é o maior contribuinte de `tjger`.

A avaliação dessas matrizes pode seguir duas linhas. Na primeira linha, caso a intenção seja buscar elementos mais participantes nas características, deve-se atentar para células com uma intensidade mais próxima ao fim da escala de cores (nesse caso, células vermelhas e amarelas). Em uma segunda linha, deve-se concentrar em células mais próximas ao início da escala (nesse caso, azuis e verdes). Em geral, elementos não participantes nas características são descartados através do recurso de filtragem da ferramenta. Os pacotes `sample.game` e `tjger.net` são exemplos disso. Pacotes específicos de algumas características são identificados buscando por linhas com poucas células coloridas, como ocorre para os pacotes `hgb.lib.xml` e `tjger.game.internal`.

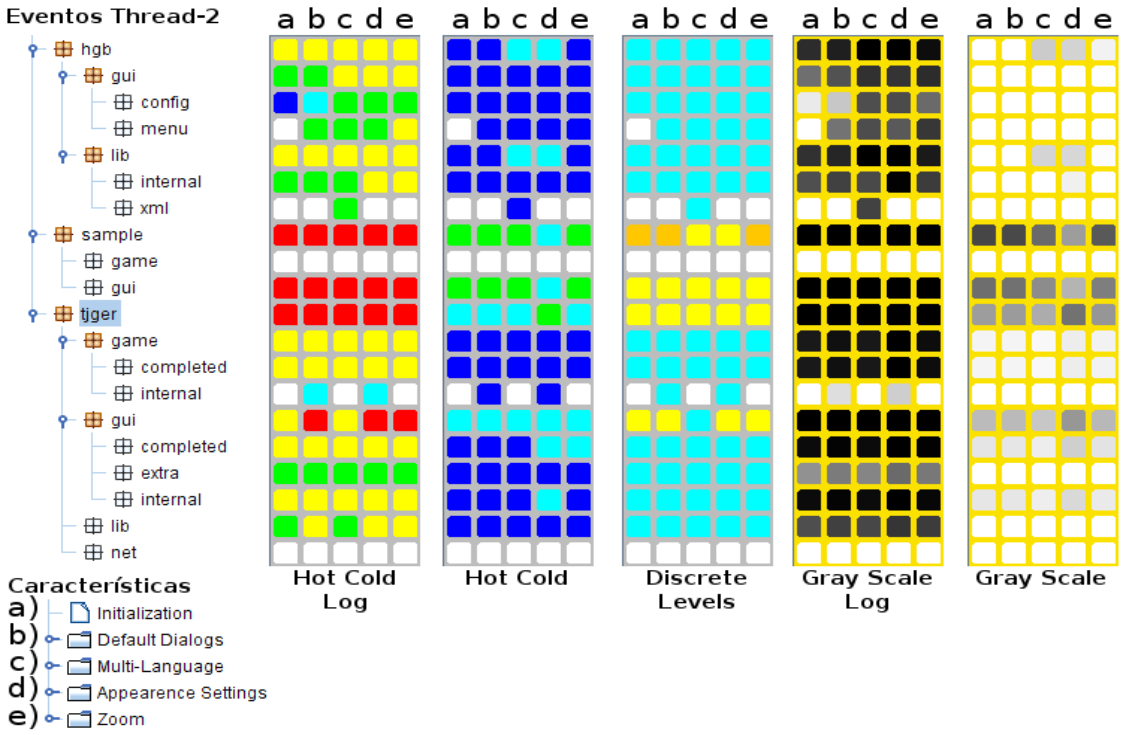


Figura 4.9: Matrizes com métrica de cobertura de característica NSEPF nas diferentes escalas de cores mostrando a representatividade dos pacotes sobre as características em termos de eventos no rastro da *Thread-2* do Roteiro 1.

A alternância entre escalas de cores é outro recurso interessante para fornecer perspectivas diferentes para uma mesma métrica. A Figura 4.9 mostra a

matriz de *Eventos* nas diferentes escalas de cores. Para a mesma métrica pode-se obter diferentes percepções sobre a participação dos elementos. Neste caso, a escala *Hot Cold Log*, a mesma usada na Figura 4.8, oferece um melhor realce das diferenças entre as métricas calculadas. A vantagem das escalas coloridas sobre as escalas de cinza está na melhor distinção entre níveis com pequena variação no valor das métricas, já que para a escala de cinza é mais difícil perceber a diferença entre dois tons muito próximos. Já as escalas logarítmicas mostram melhor distinção para métricas com grande variação em seus valores.

Finalmente, a ordenação das linhas e/ou colunas permite agrupar os elementos segundo as métricas e, conseqüentemente, segundo a intensidade de coloração das células. Isso pode facilitar a caracterização dos elementos quanto a sua participação no rastro. A Figura 4.10 ilustra as linhas ordenadas segundo o valor das métricas e a quantidade de características em que cada elemento participa. Com a ordenação das linhas, os diálogos auxiliares de linhas e colunas selecionadas sobre a matriz devem ser utilizados para orientar sobre quais elementos estão dispostos em cada linha e coluna. Pode ser observado que

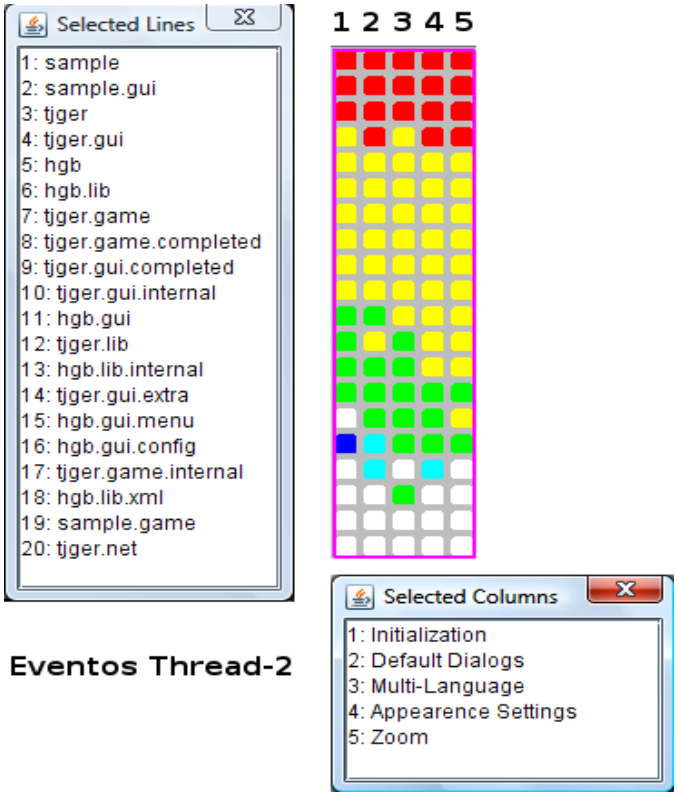


Figura 4.10: Matriz com métrica de cobertura de característica NSEPF com linhas ordenadas mostrando a representatividade dos pacotes sobre as características em termos de eventos no rastro da *Thread-2* do Roteiro 1.

elementos mais genéricos (coloração próxima ao fim da escala e participação na maior parte das características) estão nas primeiras linhas e elementos mais específicos encontram-se nas últimas linhas (coloração próxima ao início da escala e participação em poucas características).

4.6.4 Análise de Similaridade entre Elementos do Código

A identificação de quais elementos do código compartilham a execução de uma mesma característica (ou passo de execução) é o objetivo da análise de similaridade. Esta análise pode fornecer informações sobre a colaboração entre os elementos, dado que dois elementos só serão considerados similares se participam nas mesmas características ou passos de execução. A análise de similaridade é um assunto bem estabelecido tendo a disposição uma vasta literatura. Dada as condições do trabalho e considerando o volume de dados sendo tratado, a técnica *Dotplot* foi escolhida para apoiar a análise de similaridade.

Para a configuração das matrizes utilizadas nesse trabalho, o tipo de padrão *Dotplot* (*Capítulo 2 – Referencial Teórico*) esperado em um sistema bem modularizado segundo o modelo de características utilizado é o padrão *"Square"* (HELFMAN, 1996) com poucas células coloridas fora da diagonal principal ou, considerando as diferentes intensidades de coloração, com coloração próxima ao início das escalas de cores. Como é esperado um grau de compartilhamento de implementação entre as características, variações deste padrão como "*Light Cross*", "*Dark Cross*" e "*Reordered Squares*" também podem ser esperadas. Padrões do tipo "*Diagonals*" são menos propensos de serem encontrados pois estes acontecem na presença de seqüências de itens similares. No contexto deste trabalho, a medida de similaridade entre elementos do código fonte consiste na participação em características em comum. Assim, a ordem de apresentação dos elementos do código na árvore de pacotes não caracteriza necessariamente uma seqüência de itens como considerada para a análise *Dotplot* justificando a predominância de padrões "*Square*" em oposição aos padrões "*Diagonals*".

A Figura 4.11 mostra a similaridade entre os pacotes derivada dos passos de execução do Roteiro 2 (Sequência da Partida), referente ao rastro da *Thread-2* e nas diferentes escalas de cores disponíveis. A métrica utilizada foi NSFP. Dentre os padrões *Dotplot*, o único padrão observável nas matrizes é o padrão *"Light Cross"*.

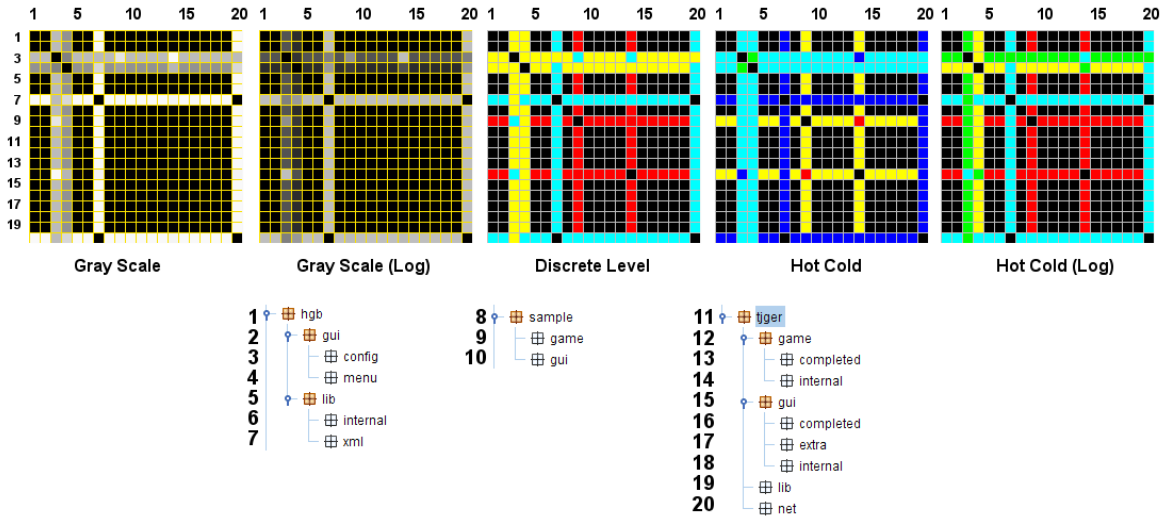


Figura 4.11: Similaridade entre os pacotes em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da *Thread-2* e na métrica NSFP.

Quatro *"cruzes"* são identificáveis em todas as matrizes: nas linhas 3, 4, 7 e 20, referentes aos pacotes `hgb.gui.config`, `hgb.gui.menu`, `hgb.lib.xml` e `tjger.net`, respectivamente. Outras duas cruzes são mais perceptíveis nas escalas coloridas, estando nas linhas 9 e 14 e referentes aos pacotes `sample.game` e `tjger.internal`. Os padrões *Dotplot* indicam uma diferença considerável desses pacotes em relação aos demais sobre suas participações nos passos de execução. Tais pacotes possuem o menor número de passos de execução compartilhados com os demais.

A intensidade de coloração das cruzes também é um indicativo do grau de compartilhamento, o que justifica o fato das linhas 9 e 14 passarem quase despercebidas nas escalas de tons de cinza, cujas diferenças de tonalidade são mais difíceis de serem percebidas. Assim, pode ser interpretado que os pacotes das linhas 7 e 20 compartilham menos passos de execução com os demais pacotes que os pacotes das linhas 9 e 14. De fato, a Tabela 4.7 resume o número de passos de execução em que cada pacote participa e confirma essas interpretações. Para este caso e em nível de pacotes, pode ser visto que os passos de execução não têm uma separação clara no nível de pacotes, já que a maior parte dos pacotes participam praticamente em todos os passos de execução. Porém, as linhas que caracterizam o padrão "Light Cross" podem indicar alguns pacotes com papel bem específico na execução do sistema, como por exemplo o pacote `hgb.lib.xml` e `tjger.net`

Tabela 4.7: Relação entre pacotes e quantidade de passos de execução compartilhados no rastro da *Thread-2* para o Roteiro 2.

	Passos Relacionados	Máximo de Passos Compartilhados
1: hgb	16	16
2: hgb.gui	16	16
3: hgb.gui.config	4	3
4: hgb.gui.menu	6	6
5: hgb.lib	16	16
6: hgb.lib.internal	16	16
7: hgb.lib.xml	1	1
8: sample	16	16
9: sample.game	14	14
10: sample.gui	16	16
11: tjger	16	16
12: tjger.game	16	16
13: tjger.game.completed	16	16
14: tjger.game.internal	13	13
15: tjger.gui	16	16
16: tjger.gui.completed	16	16
17: tjger.gui.extra	16	16
18: tjger.gui.internal	16	16
19: tjger.lib	16	16
20: tjger.net	1	1

que participam apenas no passo *"Game Menu -> New Game -> Game Configuration"* e poderiam ser alvo de uma análise mais detalhada.

A Figura 4.12 mostra a similaridade entre as classes para o pacote `sample` em relação aos passos de execução compartilhados do Roteiro 2 baseado no rastro da *Thread-2* e na métrica NSFP. De acordo com a documentação o pacote `sample` contém o código da aplicação responsável pela instanciação do framework Tjger.

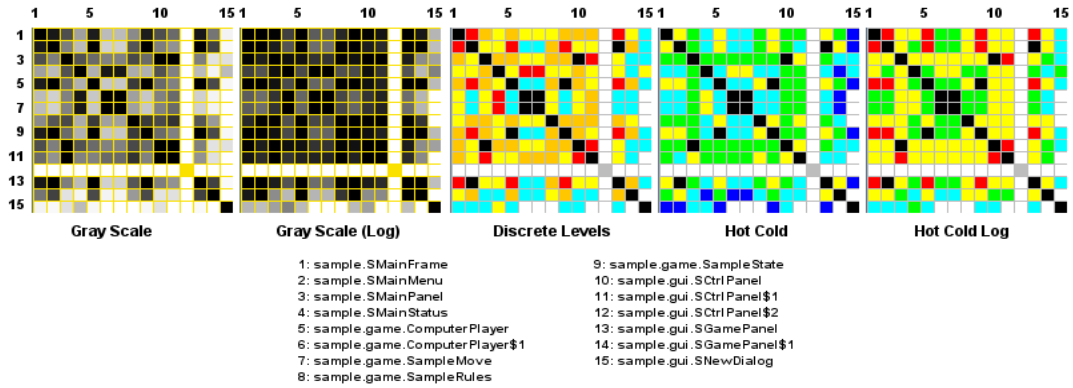


Figura 4.12: Similaridade entre classes do pacote `sample` em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da *Thread-2* e na métrica NSFP.

Novamente pode ser visto um exemplo de *"Light Cross"* na linha 12, `sample.gui.SCtrlPanel$2`, indicando que esta classe não possui eventos relacionados a nenhum passo de execução nesta *thread* (célula amarela para *Gray*

Scale Log e cinza para escala *Hot Cold Log*) e, conseqüentemente, não compartilha nenhum passo de execução com as demais classes (células brancas). O padrão é visto novamente na linha 15, `sample.gui.SNewDialog`, porém essa classe compartilha o único passo em que participa nessa *thread*, "*Game Menu -> New Game -> Game Configuration"*, com outras classes nesse pacote. Este é um forte indicativo que esta classe tem um papel bem pontual nesse passo de execução.

Examinando a diagonal principal, em especial na matriz *Hot cold Log*, pode ser visto alguns agrupamentos interessantes em função da coloração das células (preto e vermelho) nas linhas 1 e 2 (`sample.SMainFrame` e `sample.SMainMenu`), 6 e 7 (`sample.game.ComputerPlayer$1` e `sample.game.SampleMove`) e 10 e 11 (`sample.gui.SCtrlPanel` e `sample.gui.SCtrlPanel$1`). Esses pares de classes que formam "blocos" na diagonal principal indicam classes com alto grau de compartilhamento dos passos de execução.

Observando fora da diagonal principal, alguns blocos adicionais são formados, atentando-se para os pares de linhas citados anteriormente. Assim, as linhas 5, 9 e 13 (`sample.game.ComputerPlayer`, `sample.game.SampleState` e `sample.gui.SGamePlayer`) podem ser agrupadas às linhas 1 e 2, por compartilharem um valor considerável de passos de execução. De forma similar, a linha 3 (`sample.SMainPanel`) forma um segundo grupo de classes com as linhas 10 e 11. Esses agrupamentos indicam classes com participação nos mesmos passos de execução. Pela seleção das células relativas as essas classes, os passos de execução compartilhados podem ser observados.

Para o primeiro grupo (linhas 1, 2, 5, 9 e 13) as classes nas linhas 2 e 13 e nas linhas 5 e 9 tem todas os passos de execução em comum, indicado pelas células pretas nas respectivas linhas e colunas. Esses dois grupos de classes participam, respectivamente, em 16 e 14 passos de execução do Roteiro 2. A classe na linha 1 é a que participa no menor número de passos de execução (12 passos). Nesse contexto e em função do grande número de passos de execução associados, as classes desse grupo podem ser consideradas como tendo um papel mais genérico na execução do sistema.

No grupo de classes das linhas 3, 10 e 11, essas classe participam respectivamente em 7, 7 e 6 passos de execução. Curiosamente, as classe desse grupo estão associadas a passos semelhantes entre as três jogadas realizadas na

partida (*"P1 -> Click Play Card (1/2/3)" e "P1 -> Click Roll Dice (1/2/3)"*). Isso indica que as classes desse grupo têm um papel mais específico na execução. Com isso, pode ser observado que o reconhecimento desses agrupamentos de classes pode ser utilizado para indicar tanto classes com um papel mais genérico na execução, quanto classes que colaboram em uma parte específica da execução. Além disso, o reconhecimento do padrão "*Light Cross*" também pode ser útil na busca por classes com papel mais específico na execução e que tem colaboração mínima com as demais, como ocorre para a classe `sample.gui.SNewDialog`, linha 15.

Análises semelhantes às realizadas para o pacote `sample` podem ser feitas para os pacotes `tjger` e `hgb`. As figuras 4.13 e 4.14 mostram as matrizes extraídas

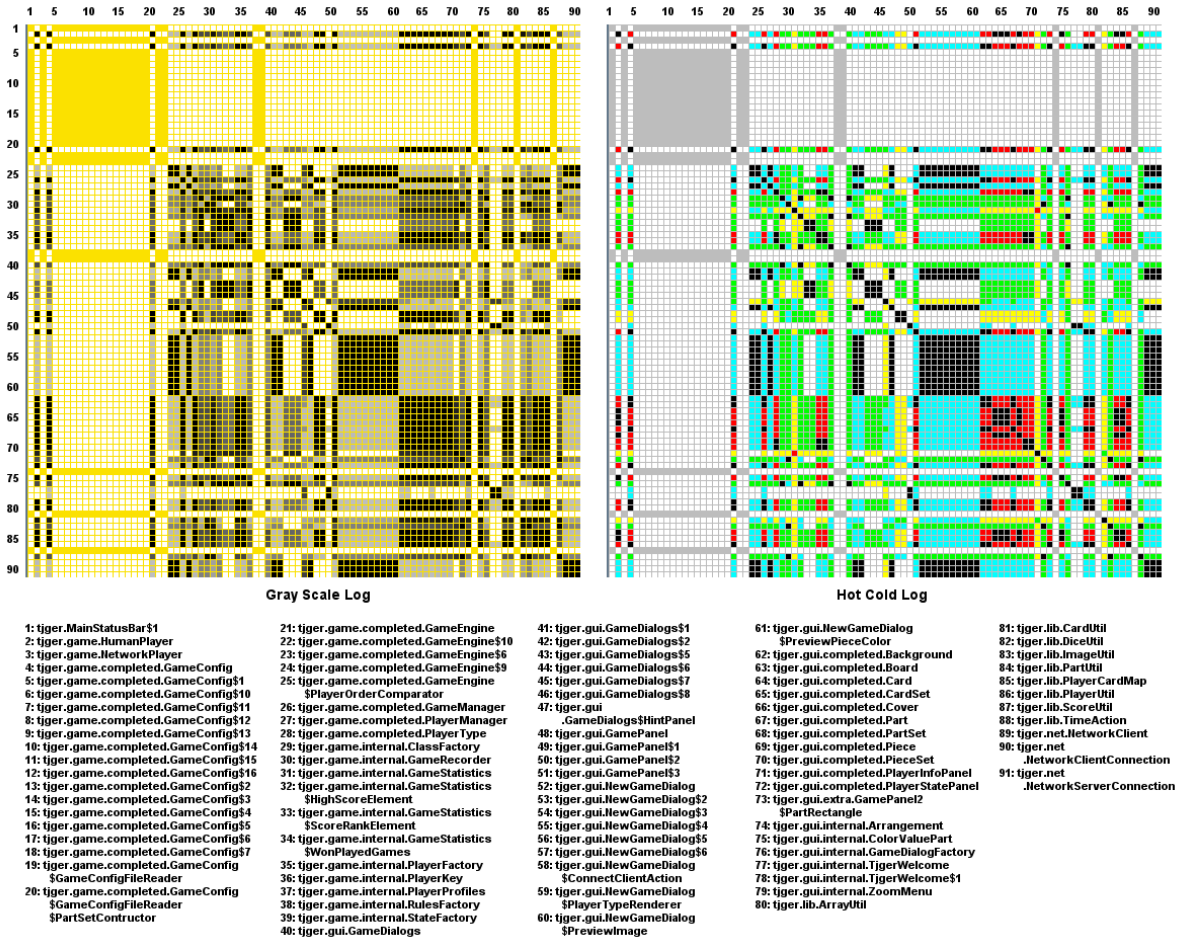


Figura 4.13: Similaridade entre classes do pacote `tjger` em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da *Thread-2* e na métrica NSFP.

para esses pacotes. Um primeira observação está na quantidade maior de classes não participantes nos passos de execução para o rastro dessa da *Thread-2* (linhas

1, 3, 5 a 20, 22, 23, 38, 39, 74, 81 e 87 na Figura 4.13 e linhas 4, 6, 20 a 26, 29, 30, 37, 38, 41 e 43 na Figura 4.14). Novamente, essas classes caracterizam o padrão "*Light Cross*". Uma das explicações para a presença dessas classes não participantes está no fato de elas terem sido carregadas, provavelmente durante a inicialização, e não terem sido utilizadas novamente durante a execução da aplicação. Como exemplo tem-se as classes `tjger.game.NetworkPlayer` (linha 3) que, caso um jogador remoto fosse habilitado, poderia ter tido um padrão similar ao de sua contraparte `tjger.game.HumanPlayer` (linha 2) a qual, segundo a documentação, guarda a implementação do jogador local na aplicação.

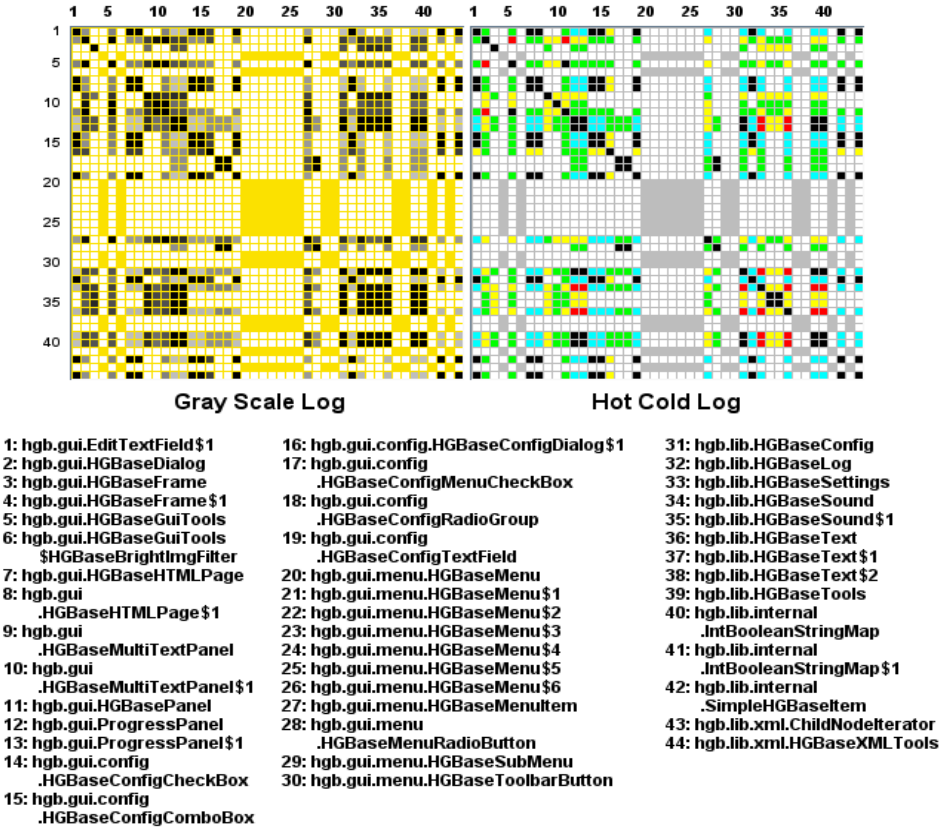
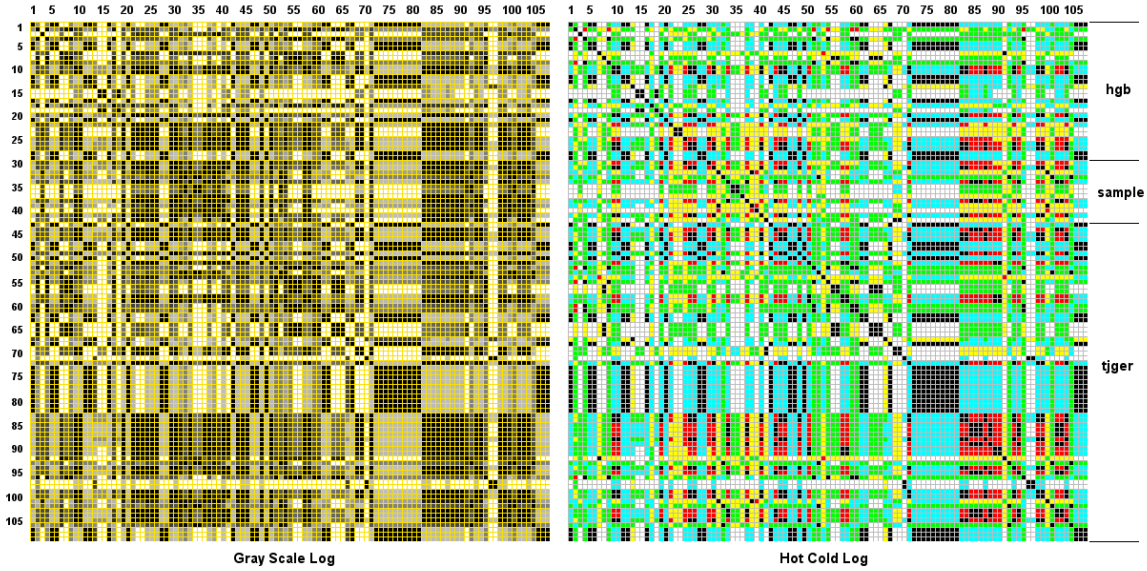


Figura 4.14: Similaridade entre classes do pacote `hgb` em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da *Thread-2* e na métrica NSFP.

Na Figura 4.13, dois blocos de classes ao longo da diagonal principal têm um destaque especial. O primeiro ocorre entre as linhas 53 e 62 e compreende um conjunto de classes internas (*inner classes*) à classe `tjger.gui.NewGameDialog`. Essas classes participam unicamente no passo *"Game Menu -> New Game -> Game Configuration"*. O mesmo se dá com as classes nas linhas 24, 25, 27, 41, 42,

47 e 89 a 91. O segundo bloco ocorre entre as linhas 63 e 88, salvo algumas classes nesse intervalo que compartilham poucos ou nenhum passo com as demais classes no bloco, como ocorre nas linhas 74, 77, 78, 81 e 87. A maioria das classes nesse bloco tem uma participação entre 13 e 16 passos de execução. As exceções ocorrem nas linhas 72 (6 passos), 73 (4), 76 (4), 82 (7), 83 (4) e 88 (4). Classes fora desse intervalo também tem um compartilhamento considerável com esse bloco de classes, o que pode ser visto na linhas 2, 4, 21, 26, 28, 35, 36 e 51. Na Figura 4.13, o compartilhamento de passos de execução entre as classes do pacote `hgb` exibe um comportamento um pouco mais disperso em relação ao pacote `tjger`. Isso pode ser verificado pela ausência de blocos contínuos indicando o compartilhamento de todos (ou da maioria) os passos de execução ao longo da diagonal principal.

Finalmente, a Figura 4.15 exibe a similaridade entre as classes dos pacotes `sample`, `tjger` e `hgb`. Classes não participantes em nenhum passo de execução do rastro da *Thread-2* foram excluídas através dos mecanismos de filtragem da ferramenta. Pode ser verificado que a maior intensidade de compartilhamento de passos de execução entre `sample` e `tjger` ocorre entre as linhas 83 a 96 e 99 a 105. O mesmo pode ser observado entre `hgb` e `tjger`. Entre `sample` e `hgb` ocorre entre as linhas 22 a 27. Isso mostra que as classes entre esses pacotes não participam isoladamente em cada passo de execução e provavelmente colaboram na implementação. Maiores detalhes dos passos de execução compartilhados entre classes de diferentes pacotes podem ser obtidos pela seleção das respectivas células.



1: hgb.gui.EditTextField$1
2: hgb.gui.HGBaseDialog
3: hgb.gui.HGBaseFrame
4: hgb.gui.HGBaseGuiTools
5: hgb.gui.HGBaseHTMLPage
6: hgb.gui.HGBaseHTMLPage$1
7: hgb.gui.HGBaseMultiTextPanel
8: hgb.gui.HGBaseMultiTextPanel$1
9: hgb.gui.HGBasePanel
10: hgb.gui.ProgressPanel
11: hgb.gui.ProgressPanel$1
12: hgb.gui.config.HGBaseConfigCheckBox
13: hgb.gui.config.HGBaseConfigComboBox
14: hgb.gui.config.HGBaseConfigDialog$1
15: hgb.gui.config .HGBaseConfigMenuCheckBox
16: hgb.gui.config.HGBaseConfigRadioGroup
17: hgb.gui.config.HGBaseConfigTextField
18: hgb.gui.menu.HGBaseMenuItem
19: hgb.gui.menu.HGBaseMenuRadioButton
20: hgb.lib.HGBaseConfig
21: hgb.lib.HGBaseLog
22: hgb.lib.HGBaseSettings
23: hgb.lib.HGBaseSound
24: hgb.lib.HGBaseSound$1
25: hgb.lib.HGBaseText
26: hgb.lib.HGBaseTools
27: hgb.lib.internal.IntBooleanStringMap
28: hgb.lib.internal.SimpleHGBaseItem
29: hgb.lib.xml.HGBaseXMLTools
30: sample.SMainFrame
31: sample.SMainMenu
32: sample.SMainPanel
33: sample.SMainStatus
34: sample.game.ComputerPlayer
35: sample.game.ComputerPlayer$1
36: sample.game.SampleMove
37: sample.game.SampleRules
38: sample.game.SampleState
39: sample.gui.SCtrlPanel
40: sample.gui.SCtrlPanel$1
41: sample.gui.SGamePanel
42: sample.gui.SGamePanel$1
43: sample.gui.SNewDialog
44: tjger.game.HumanPlayer
45: tjger.game.completed.GameConfig
46: tjger.game.completed.GameEngine
47: tjger.game.completed.GameEngine$9
48: tjger.game.completed.GameEngine $PlayerOrderComparator
49: tjger.game.completed.GameManager
50: tjger.game.completed.PlayerManager
51: tjger.game.completed.PlayerType
52: tjger.game.internal.ClassFactory
53: tjger.game.internal.GameRecorder
54: tjger.game.internal.GameStatistics
55: tjger.game.internal.GameStatistics $HighScoreElement
56: tjger.game.internal.GameStatistics $ScoreRankElement
57: tjger.game.internal.GameStatistics $WonPlayedGames
58: tjger.game.internal.PlayerFactory
59: tjger.game.internal.PlayerKey
60: tjger.game.internal.PlayerProfiles
61: tjger.gui.GameDialogs
62: tjger.gui.GameDialogs$1
63: tjger.gui.GameDialogs$2
64: tjger.gui.GameDialogs$5
65: tjger.gui.GameDialogs$6
66: tjger.gui.GameDialogs$7
67: tjger.gui.GameDialogs$8
68: tjger.gui.GameDialogs $HintPanel
69: tjger.gui.GamePanel
70: tjger.gui.GamePanel$1
71: tjger.gui.GamePanel$2
72: tjger.gui.GamePanel$3
73: tjger.gui.NewGameDialog
74: tjger.gui.NewGameDialog$2
75: tjger.gui.NewGameDialog$3
76: tjger.gui.NewGameDialog$4
77: tjger.gui.NewGameDialog$5
78: tjger.gui.NewGameDialog$6
79: tjger.gui.NewGameDialog $ConnectClientAction
80: tjger.gui.NewGameDialog $PlayerTypeRenderer
81: tjger.gui.NewGameDialog $PreviewImage
82: tjger.gui.NewGameDialog $PreviewPieceColor
83: tjger.gui.completed.Background
84: tjger.gui.completed.Board
85: tjger.gui.completed.Card
86: tjger.gui.completed.CardSet
87: tjger.gui.completed.Cover
88: tjger.gui.completed.Part
89: tjger.gui.completed.PartSet
90: tjger.gui.completed.Piece
91: tjger.gui.completed.PieceSet
92: tjger.gui.completed.PlayerInfoPanel
93: tjger.gui.completed.PlayerStatePanel
94: tjger.gui.extra.GamePanel2 $PartRectangle
95: tjger.gui.internal.ColorValuePart
96: tjger.gui.internal.GameDialogFactory
97: tjger.gui.internal.TjgerWelcome
98: tjger.gui.internal.TjgerWelcome$1
99: tjger.gui.internal.ZoomMenu
100: tjger.lib.ArrayUtil
101: tjger.lib.DiceUtil
102: tjger.lib.ImageUtil
103: tjger.lib.PartUtil
104: tjger.lib.PlayerCardMap
105: tjger.lib.PlayerUtil
106: tjger.lib.TimeAction
107: tjger.net.NetworkClient
108: tjger.net.NetworkClientConnection
109: tjger.net.NetworkServerConnection

Figura 4.15: Similaridade entre classes dos pacote `hgb`, `sample` e `tjger` em relação aos passos de execução compartilhados do Roteiro de Execução 2, baseado no rastro da *Thread-2* e na métrica NSFP.

4.6.5 Análise de Similaridade de Implementação entre Características

A similaridade entre as características (e passos de execução) também pode ser analisada. A finalidade dessa análise é mostrar se há ou não compartilhamento de elementos do código entre as características, obviamente em relação à ativação dos elementos durante a execução. Isto pode servir como um indicativa de reuso no sistema para na implementação de características diferentes. Analogamente aos elementos do código, recorre-se à técnica *Dotplot* para apoiar a na análise de similaridade entre as características.

A análise de similaridade entre as características e os passos de execução

segue a mesma abordagem de busca por padrões *Dotplot* utilizada na seção anterior. Porém, considerando que os passos de execução têm uma ordem de execução específica é possível que alguns padrões do tipo "*Diagonals*" possam ser encontrados, uma vez que ordem de execução do roteiro seja mantida na disposição de linhas e colunas das matrizes.

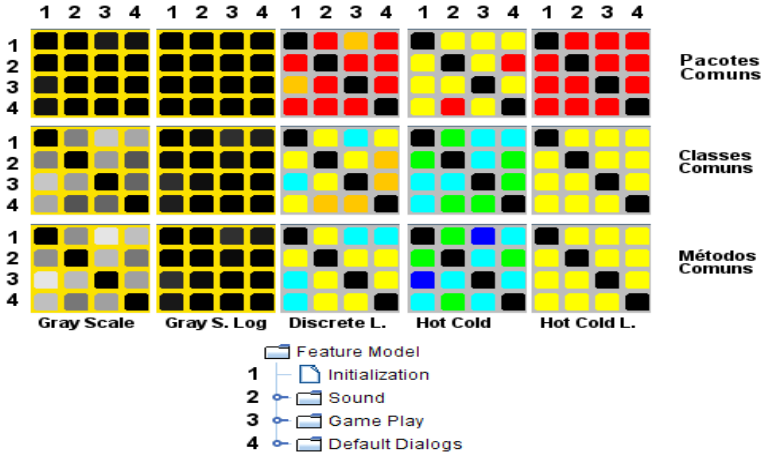


Figura 4.16: Similaridade entre características quanto ao número de pacotes (NSPF), classe (NSCF) e métodos (NSMF) compartilhados durante a execução do Roteiro 2 e para os rastros da *Thread-2*.

A Figura 4.16 mostra o primeiro nível de similaridade que pode ser encontrado entre as características para o rastro de execução gerado pela *Thread-2* do Roteiro 2. O número de pacotes (NSFP), classes (NSFC) e métodos (NSFM) é levado em conta para a geração das matrizes em cada escala. No nível de pacotes (primeira "linha" de matrizes), praticamente todos os pacotes são compartilhados pelas características. No nível de classe, essa relação começa a se afrouxar. Finalmente no nível de métodos, começa a haver uma distinção maior entre as características. Porém, com essa pequena quantidade de características é difícil extrair qualquer informação mais relevante das matrizes. Uma alternativa é expandir as características até o nível do passos de execução. A Figura 4.17 ilustra essa expansão e apresenta as matrizes resultantes contendo nas linhas e colunas apenas os passos de execução do Roteiro 2.

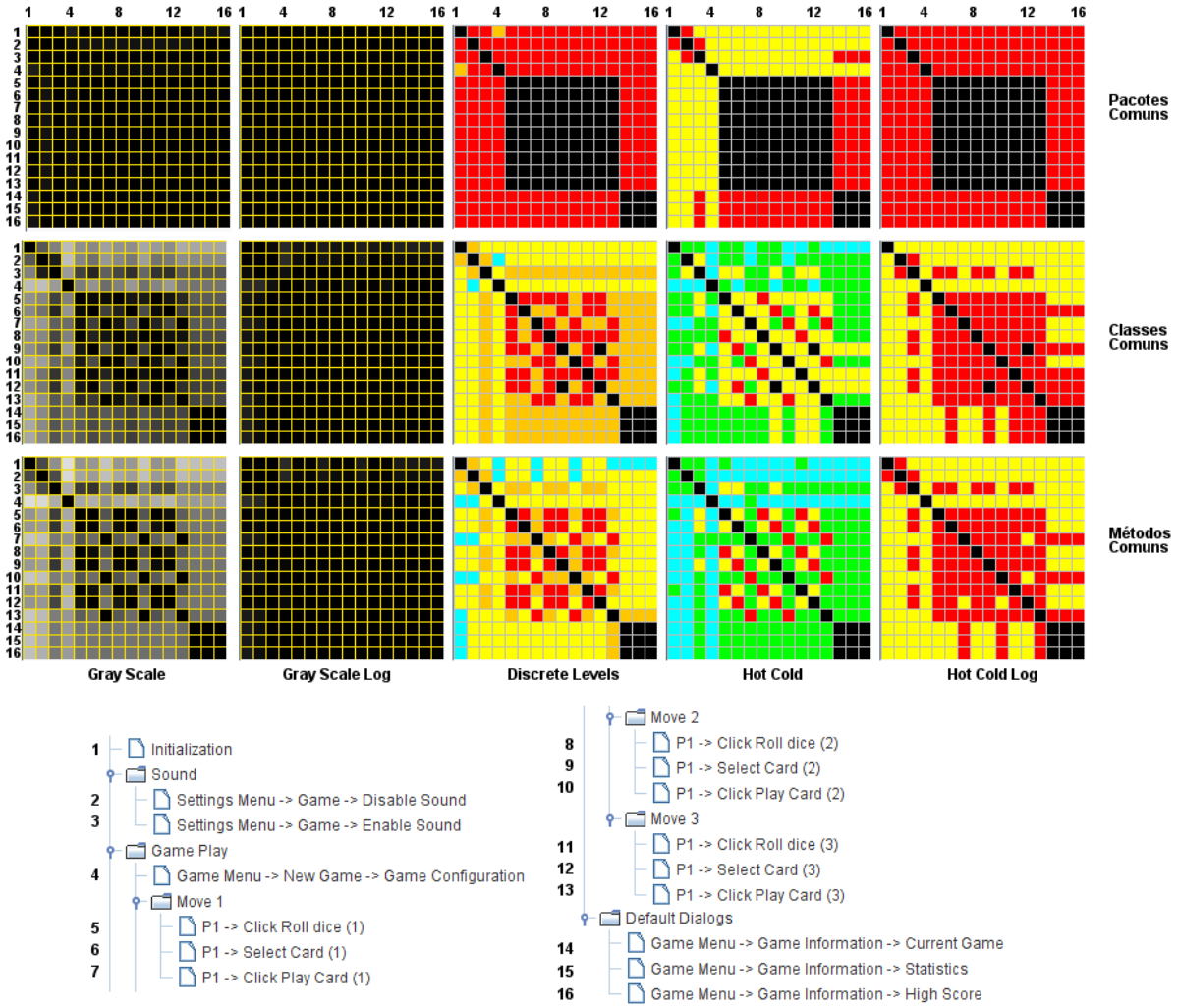


Figura 4.17: Similaridade entre os passos de execução quanto ao número de pacotes (NSPF), classe (NSCF) e métodos (NSMF) compartilhados durante a execução do Roteiro 2 e para os rastros da *Thread-2*.

Nas matrizes de pacotes comuns, em especial na matrizes coloridas, os blocos de células pretas ao longo da diagonal principal deixam clara a separação entre os passos de execução 5 a 13, que representam as jogadas da partida, e 14 a 16, que representam a exibição dos diálogos de informações sobre a partida. As matrizes de classes comuns e métodos comuns dão uma noção mais refinada da relação entre os passos de execução. É possível ver neste caso a replicação dos passos de execução que compõem as três jogadas da partida (“Move 1/2/3”).

Na Figura 4.18, os passos referentes a cada jogada estão destacados em azul, ao longo da diagonal principal. As células em vermelho representam a replicação dos passos, podendo ser vista também como um exemplo do padrão “*Diagonals*”. Neste caso, a interpretação para o padrão seria de que cada passo na seqüência (jogada) executa praticamente os mesmos métodos que os respectivos

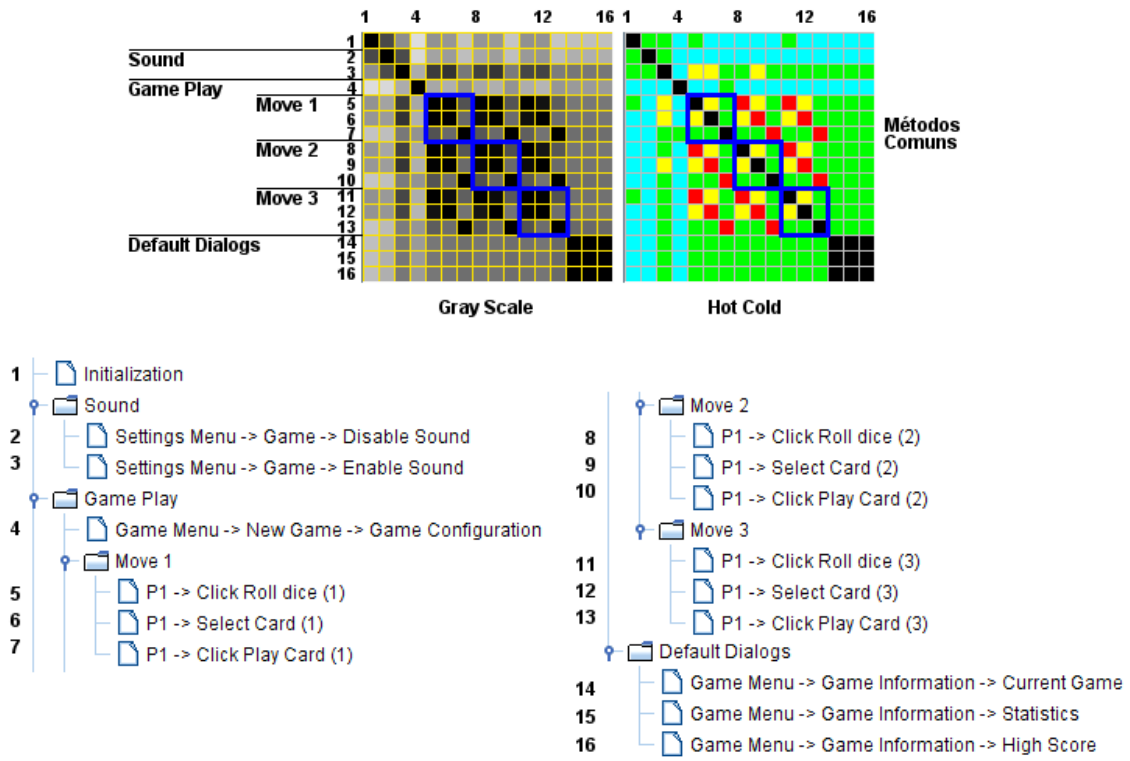


Figura 4.18: Similaridade entre passos de execução (NSMF) com destaque para a replicação das jogadas na partida para o Roteiro 2 formando o padrão *Dotplot Diagonals*.

passos nas seqüências seguintes.

5 Considerações Finais

Neste capítulo a abordagem proposta para a análise das características foi apresentada. Cinco formas de análise visual foram descritas. A abordagem segue etapas relativamente simples e diretas. As análises visuais porém ainda podem consumir tempo e esforço considerável, em especial, quando os objetivos a serem alcançados não estão bem estabelecidos. A possibilidade de se analisar tanto elementos mais específicos (relacionados a poucas características) quanto elementos mais gerais (relacionados às várias características) é um dos diferenciais da abordagem, possibilitado especialmente pelo uso de escalas de cores combinadas às matrizes de visualização. Outro ponto é a possibilidade de se lidar com várias características ao mesmo tempo com base em um único rastro de execução. Essas podem ainda ser refinadas e organizadas em uma hierarquia com vários níveis de granularidade. O estudo apresentado no Capítulo 5 foi conduzido

com base nessas abordagens de análises.

A escalabilidade é uma questão relevante e que ainda merece estudos mais aprofundados. Porém, é importante destacar o que foi feito até o momento sobre essa questão. Do ponto de vista de processamento e espaço, três estratégias foram adotadas: 1. processamento sobre demanda , dado que os cálculos são realizados apenas para elementos visualizados nas matrizes; 2. indexação dos elementos do código e características quanto a sua localização no rastro de execução; 3. armazenamento dos cálculos realizados em cache. Quanto a escalabilidade na visualização, as opções utilizadas foram: 1. a navegação iterativa sobre os modelos de código fonte e características; 2. o redimensionamento dinâmico das células nas matrizes de visualização (Zoom); 3. a filtragem de elementos nas linhas e colunas. Finalmente, a quantidade de roteiros de execução necessárias para cobrir uma grande quantidade de características no sistema pode ser uma ameça à escalabilidade da abordagem. Porém, o intuito não está em fornecer uma cobertura exaustiva de características. O objetivo principal é fornecer pontos de partida para análises mais detalhadas de características específicas e dos elementos do código correspondentes.

CAPÍTULO 5. ANÁLISE DO ARGOUML

1 OBJETIVOS GERAIS

Neste capítulo o estudo de algumas características de uma aplicação de grande porte será realizado. A ferramenta CASE de modelagem UML, *ArgoUML 0.19.3*, será utilizada como alvo da análise. O estudo tem como objetivo principal testar a ferramenta *Featincode* em um sistema de grande porte. Para isso, dois contextos foram escolhidos: o primeiro considerando características similares da aplicação, o segundo considerando características distintas e com pontos em comuns.

Na Seção 2, o sistema alvo, ArgoUML, é descrito incluindo motivações para a escolha, características e uma breve descrição de sua interface gráfica. Seção 3 apresenta o estudo feito com o Contexto 1 (características similares). Seção 4 mostra o estudo do Contexto 2 (características distintas com pontos em comum). Seção 5 discute problemas e limitações da abordagem encontradas durante o estudo. Seção 6 relata as considerações finais.

2 DESCRIÇÃO DO SISTEMA ALVO

ArgoUML é uma ferramenta de modelagem UML que possui suporte para o diagramas UML padrão 1.4. É essencialmente escrita em Java, podendo assim ser rodada em qualquer plataforma compatível. Em suas versões mais recentes (TIGRIS.ORG, 2008), está disponível em 10 línguas e dentre suas características distintas estão funcionalidades de suporte cognitivo, tais como: geração de críticas de projeto, automação de correções, lista de tarefas/verificação e utilização de um modelo de usuário. A versão utilizada nos experimentos é a versão *0.19.3*, por ter sido esta a versão utilizada e preparada no início da pesquisa. Porém, a abordagem é igualmente aplicável às versões mais recentes.

2.1 MOTIVAÇÕES

Várias razões levaram à escolha de *ArgoUML,* dentre elas: o porte do sistema, a disponibilidade de documentação de usuário e de projeto, a disponibilidade do código fonte e o fato de ser uma ferramenta utilizada para testes em outros estudos na área, por exemplo, em (GREEVY; DUCASSE, 2005b) e (GREEVY, 2007).

ArgoUML pode ser considerado um sistema de grande porte em função da grande quantidade de elementos envolvidos em seu código fonte. A Tabela 5.1 mostra algumas métricas extraídas da versão 0.19.3 com o plugin *Metrics* (SAUER, 2008) para o Eclipse. As métricas extraídas são: 1. NOP: número de pacotes (com classes ou interfaces); 3. NOC: número de classes; 4. NOI: número de interfaces; 5. NOM: número de métodos; 6. NOC+NOI: número de classes + interfaces.

Tabela 5.1: Métricas extraídas do código fonte do *ArgoUML* 0.19.3 com o plugin do Eclipse PMD.

	NOP	NOC	NOI	NOM	NOC+NOI
org.argouml	71	1229	66	8297	1295
org.tigris.gef	19	276	28	2934	304
org.tigris.toolbar	3	31	1	221	32
ru.novosoft.uml	15	146	103	3676	249
Sub-total	108	1682	198	15128	1880
Outros	30	233	72	2633	305
Total	138	1915	270	17761	2185

O projeto surgiu de uma iniciativa de *Jason Robbins* na *Universidade de Irvine na Califórnia (UCI)* em criar um ambiente de desenvolvimento com ênfase em fatores humanos. Como iniciativa de pesquisa, o sistema recebeu a colaboração de diversos pesquisadores e desenvolvedores. O sistema possui extensa documentação de usuário e de projeto, especialmente pela colaboração da comunidade e dos desenvolvedores. Além disso, o código fonte é aberto sobre a licença BSD. Atualmente, a principal fonte de acesso ao projeto é o *website* da *Tigris.org*, uma comunidade de código fonte aberto focada na produção de ferramentas para o desenvolvimento colaborativo de software (TIGRIS.ORG, 2008). Dessa maneira, a reprodução e validação dos experimentos realizados torna-se mais fácil para outros pesquisadores.

2.2 CARACTERÍSTICAS

Além das características tradicionais de outras ferramentas de modelagem UML (criação/edição/armazenamento de diagramas, exportação em diferentes formatos de imagens, opções avançadas de edição), *ArgoUML* traz uma série de características relativas ao suporte cognitivo que refletem sua proposta fundamental. A Figura 5.1 mostra um digrama de alto nível das principais características do *ArgoUML*, disponíveis em sua documentação e no *website*. Após examinar a documentação e experimentar a ferramenta, é possível refinar essas características e obter diagramas mais detalhados como o da Figura 5.2. Os roteiros de execução para o estudos neste capítulo foram definidos, partindo desses diagramas.

Figura 5.1: Modelo de Características com principais características do *ArgoUML*.

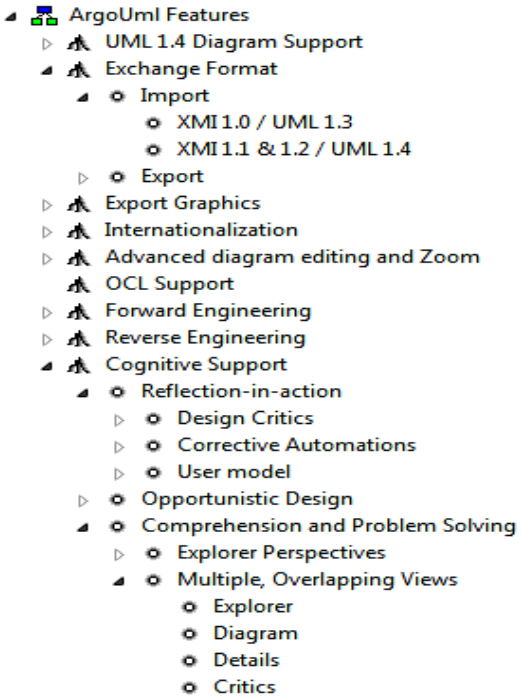


Figura 5.2: Modelo de Características do *ArgoUML* com algumas expansões nos nós principais.

2.3 Breve Descrição da Interface Gráfica do ArgoUML

A Figura 5.3 mostra a janela principal do *ArgoUML*. Há nela 4 painéis mais importantes: 1. o *"Explorer Pane"* (área 5) é responsável por dar acesso a todos os elementos criados no projeto; 2. o *"Editor Pane"* (área 9) permite a edição dos diagramas UML 1.4; 3. o *"To-Do Pane"* (área 7) contém a lista de tarefas do projeto, em especial, as tarefas geradas automaticamente pelo *ArgoUML*; 4. o *"Details Pane"* (área 10), que mostra informações detalhadas sobre qualquer elemento do modelo selecionado.

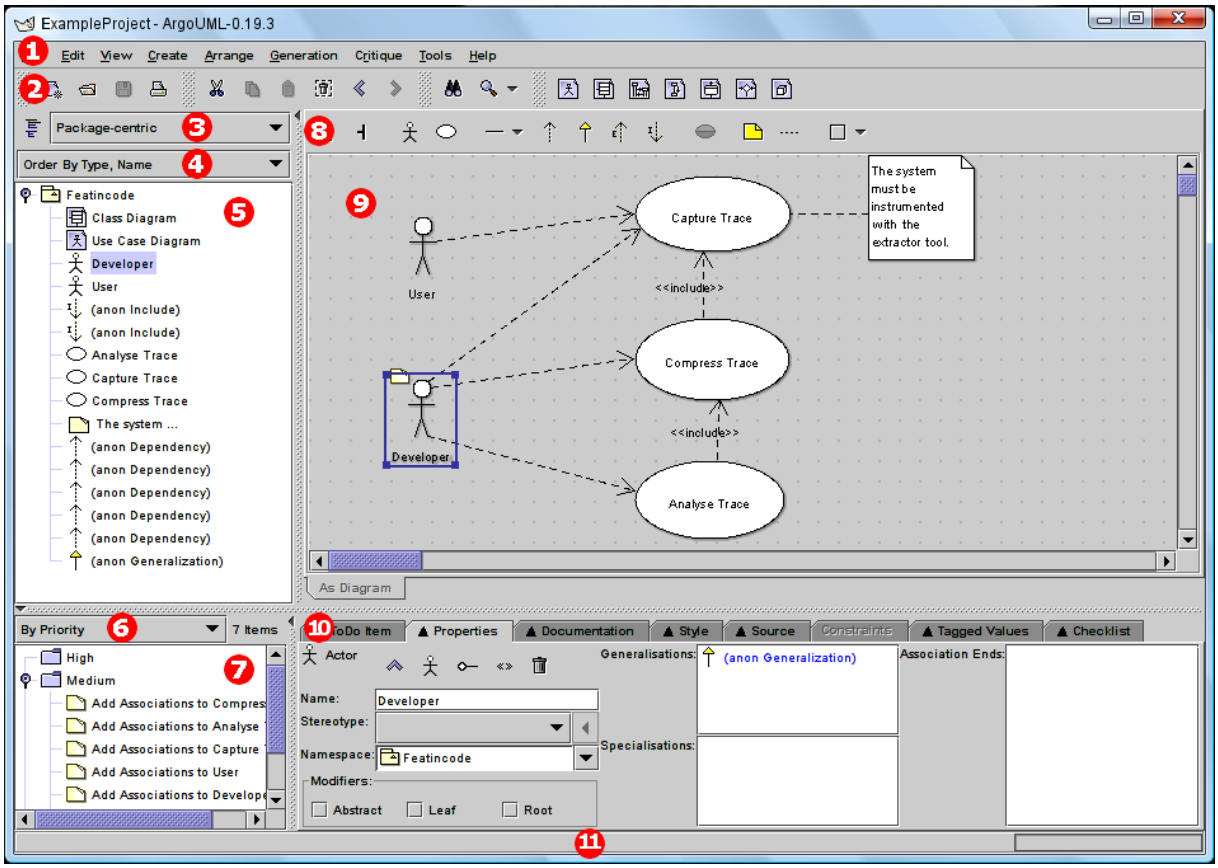


Figura 5.3: Janela principal do *ArgoUML-0.19.3*.

A seguir os demais itens da interface são enumerados e brevemente descritos, conforme os rótulos inseridos na Figura 5.3. Maiores detalhes podem ser encontrados na documentação oficial da ferramenta (TIGRIS.ORG, 2008).

1. *Barra de Menu*: dá acesso a todas as funcionalidades da ferramenta;
2. *Barra de Ferramentas*: permite acessar diretamente as principais funcionalidades da ferramenta;

3. *Seleção de Perspectiva*: alterna entre as diferentes perspectivas de visualização dos elementos no *"Explorer Pane" (Package-Centric, Class-Centric, Diagram-Centric, Inheritance-Centric, Class-Associations, Residence-Centric, State-Centric e Transition-Cetric)*.
4. *Seleção de Ordenação dos Elementos*: alterna entre as diferentes formas de ordenação dos elementos no *"Explorer Pane"*.
5. *"Explorer Pane"*: dá acesso aos elementos do modelo em uma visualização em árvore.
6. *Seleção de Ordenação das Tarefas*: alterna a forma de ordenação das tarefas no *"To-Do Pane"*.
7. *"To-Do Pane"*: mostra as tarefas dos projeto geradas manualmente, pelo usuário, ou automaticamente, pela ferramenta através de seu sistema de críticas.
8. *Barra de Ferramentas do "Editor Pane"*: dá acesso as principais funcionalidades do editor de diagramas, sendo específica para cada tipo de diagrama.
9. *"Editor Pane"*: permite a edição dos diferentes diagramas UML suportados.
10. *"Details Pane"*: mostra informações detalhadas e específicas de cada elemento selecionado no projeto, permitindo editar suas propriedades.
11. *Barra de Status*: mostra pequenos avisos, por exemplo, mensagens de erro relacionadas a edição do diagrama.

3 Contexto 1: Características Muito Similares

Seção 3.1 traz os objetivos específicos para o contexto 1, que trata de características muito similares. Seção 3.2 descreve a seqüência dos roteiros de execução e na Seção 3.3 é mostrada a análise feita com relação a este contexto e os resultados obtidos. As discussões sobre este contexto são apresentadas na Seção 3.4.

3.1 Objetivos específicos

Neste primeiro contexto, pretende-se destacar elementos comuns e distintos entre roteiros de execução similares, além de inferir a semântica de eventos e *threads* do rastro. Também será mostrado como a inicialização e as características que executam em "plano de fundo" durante a execução do sistema podem influenciar a análise. Finalmente, apresenta-se a redução na quantidade de elementos a serem analisados partindo do total de elementos no código e chegando aos elementos associados com cada característica de interesse. Deseja-se com isso avaliar a capacidade da ferramenta em auxiliar essas atividades.

3.2 Descrição dos Roteiros de Execução

Existem várias maneiras de criar um mesmo diagrama com o *ArgoUML*. É esperado que independentemente do tipo de interação ou seqüência de passos que o usuário escolha para realizar tal operação, vários elementos do código sejam comuns. Assim, os testes propostos consistem na criação de diagramas de classe usando as diferentes formas de interação oferecidas pelo *ArgoUML*. A Tabela 5.2 resume os pontos similares e distintos nas formas de interação utilizadas no experimento.

Tabela 5.2. Pontos similares e distintos entre os formas de interação para criação de diagramas de classe no *ArgoUML*.

Similar	***Distintos***
● Criar diagrama de classes ○ Uma nova entrada é criada na área "Explorer Pane"; ○ O "Editor Pane" é atualizado com as ferramentas correspondentes para o diagrama criado; ○ O "Details Pane" é atualizado com as opções disponíveis para o diagrama criado.	● Menu ○ Menu de opções exibido para o usuário; ○ Opção desejada deve ser selecionada (mouse). ● Atalhos ○ Menu de opções exibido para o usuário; ○ Opção desejada deve ser selecionada (teclado). ● Barra de ferramentas ○ Seleção direta pela barra de ferramentas.

Em resumo, os passos de execução consistem na criação de diagramas de classes através do *menu*, de *atalhos de teclado* e do *botão na barra de ferramentas*. Essa ordem define a *Seqüência 1.* As Seqüência *2* a *4* são variações sobre a ordem da *Seqüência 1* e foram realizadas para verificar se havia alguma influência na ordem de execução para a análise. Antes de cada uma dessas seqüências, os passos definidos na *Seqüência Prévia* são realizados para interromper a execução do sistema de auto-críticas do *ArgoUML*. Após alguns experimentos, foi observado que, por padrão, as críticas automáticas ficavam em execução continuamente durante a execução do sistema gerando eventos não necessariamente associado às características exercitadas, assim optou-se por desabilitá-las. Para mostrar a influência da inicialização e do sistema de auto-críticas do *ArgoUML*, três roteiros adicionais foram realizados, respectivamente, *Inicialização*, *Auto-Críticas 1* e *2*. A seguir os passos de execução realizados serão descritos em detalhes. Com o objetivo de reduzir a ocorrência de eventos arbitrários, foram realizadas mais de uma execução para cada seqüência de passos. Na ocorrência de grandes discrepâncias, em especial, no número de eventos, a captura dos dados era refeita para a referida seqüência.

3.2.1 Seqüência prévia para as seqüências 1 a 4:

1 - Aguardar carregamento completo do *ArgoUML*.

2 - Desabilitar críticas automáticas do *ArgoUML*.

a - Clicar sobre o menu “Critiques”

b - Desmarcar a opção “Toogle Auto-Critique”

3.2.2 Seqüência 1:

1 - Criar Diagrama de Classes pelo menu (***menu***)

a - Clicar no menu Create

b - Clicar na opção *"New Class Diagram"*

2 - Criar Diagrama de Classes por atalhos de teclado (***shortcut***)

a - Pressionar <ALT> + C (atalho para o menu Create)

b - Posicionar a seleção sobre a opção *"New Class Diagram"* usando as setas do teclado

c - Pressionar <ENTER>

3 - Criar Diagrama de Classes pela barra de ferramentas (***toolbar***)

a - Clicar sobre o botão correspondente para a criação de diagramas de classe na barra de ferramentas

3.2.3 Sequência 2:

Idem *Sequência 1* mudando a ordem para 2, 3, 1.

3.2.4 Sequência 3:

Idem *Sequência 1* mudando a ordem para 3, 1, 2.

3.2.5 Sequência 4:

Idem *Sequência 1* mudando a ordem para 3, 2, 1.

3.2.6 Inicialização:

Iniciar o *ArgoUML* aceitando a opção de marcação da inicialização. Após o carregamento e a apresentação da janela principal do sistema, encerrar a marcação da inicialização e sair do programa.

3.2.7 Auto-Críticas 1:

Após a inicialização, desabilitar o sistema de críticas do *ArgoUML* como descrito anteriormente. Aguardar 10 segundos. Encerrar a execução.

3.2.8 Auto-Críticas 2:

Idem anterior, mas dessa vez sem desabilitar o sistema de críticas e aguardando 15 segundos para encerrar a execução (para compensar o tempo gasto para desabilitar as críticas na execução anterior).

3.3 ANÁLISES E RESULTADOS

As análises realizadas com base nos roteiros de execução para o Contexto 1 serão descritas nessa seção. Seção 3.3.1 resume o total de eventos gerados por cada *thread* em cada um das execuções. As análises definidas no capítulo de metodologias serão mostradas das Seção 3.3.2 a 3.3.6. Seção 3.3.7 mostra a redução de escopo para análise e compreensão do sistema.

3.3.1 Dados Coletados

A Tabela 5.3 mostra o resumo das execuções realizadas. Nela é mostrado o número de eventos (chamadas de métodos) gerado por cada uma das *threads* iniciadas durante cada execução do *ArgoUML* e capturados pela ferramenta de instrumentação.

Tabela 5.3: Número de eventos gerados no rastro para as execuções relacionadas ao primeiro contexto: Características Muito Similares.

		Thread-1	Thread-2	Thread-3	Thread-4	Thread-5
Sequência 1	*Execução 1*	100.255	57.580	204	15.933	7.053
	Execução 2	100.255	61.052	65	15.933	2.527
	Execução 3	100.255	44.916	379	15.933	13.842
Sequência 2	*Execução 1*	100.255	47.197	228	15.933	4.246
	Execução 2	100.255	44.677	41	15.933	4.790
	Execução 3	100.264	47.667	403	15.933	11.579
Sequência 3	*Execução 1*	100.255	44.637	216	15.933	7.053
	Execução 2	100.255	45.676	566	15.933	18.368
	Execução 3	100.255	44.738	216	15.933	4.790
Sequência 4	*Execução 1*	100.255	44.528	29	15.933	2.527
	Execução 2	100.255	44.851	29	15.933	2.527
	Execução 3	100.255	45.584	65	15.933	1.734
Inicialização	*Execução 1*	100.282	2.810	216	15.933	7.053
	Execução 2	100.282	1.754	204	15.933	9.316
Auto-Crítica 1	*Execução 1*	100.282	5.580	216	15.933	11.579
	Execução 2	100.282	8.126	403	15.933	16.105
Auto-Crítica 2	*Execução 1*	100.282	7.043	1.441	15.933	52.313
	Execução 2	100.282	672	1.266	15.933	45.524

É importante destacar que a execução das inicializações foi feita dias após a execução das seqüências 1 a 4. Assim, a diferença no número de eventos (especialmente para *Thread-1*) pode ser devido a alguma configuração utilizada na inicialização do *ArgoUML* que não correspondia exatamente a utilizada na execução

das seqüências. A execução das inicializações não era planejada inicialmente e foi feita para confirmar que alguns eventos e *threads* ocorriam exclusivamente na inicialização do sistema. O mesmo pode ser dito sobre as execuções de “Auto-Críticas 1” e “Auto-Críticas 2”, que foram realizadas para mostrar que as Threads 3 e 5 estavam associadas ao sistema de auto críticas do *ArgoUML*.

3.3.2 Análise do Espalhamento das Características pelo Rastro

A análise do espalhamento das características foi dividida em três partes. Os eventos gerados pela inicialização são analisados na Seção 3.3.2.1. A influência das autocríticas é analisada na Seção 3.3.2.2. Por fim, os eventos relacionados às características de interesse são analisados na Seção 3.3.2.3.

3.3.2.1 Inicialização

Em todos os casos foi observada a execução de cinco *threads*. Pela similaridade na quantidade de eventos geradas por cada *thread* entre as execuções percebe-se que as *threads* eram disparadas sempre na mesma ordem, o que pode ser um indício de que a função de cada uma delas possa ser fixa (Tabela 5.3). Outro detalhe que chama atenção é no número fixo de eventos para as *threads* 1 e 4 em todas as execuções. Isto sugere que os eventos dessas *threads* são sempre executados independente da interação do usuário com a ferramenta. Pela execução exclusiva das inicializações pode-se constatar que as *threads* 1 e 4 estão realmente associadas apenas à inicialização do sistema como mostra a Figura 5.4.

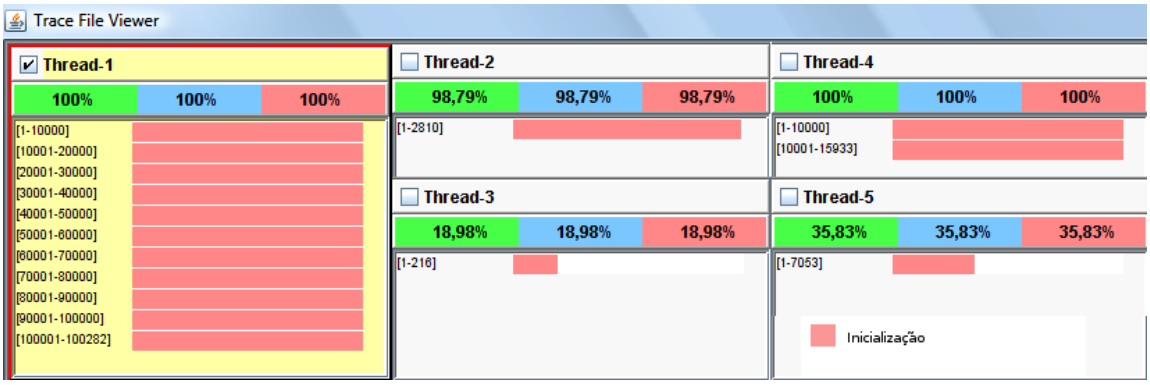


Figura 5.4: Eventos associados à Execução 1 da Inicialização do *ArgoUML*.

3.3.2.2 Auto-Críticas

As *threads* 3 e 5 estão associadas à geração de Críticas pelo *ArgoUML*, como pode ser observado na Figura 5.5 e 5.6. A espera na Figura 5.5 está associada apenas a um pequeno percentual da *Thread-2*, possivelmente pelos eventos de atualização da interface associados a essa *thread*. Na Figura 5.6Erro: Origem da referência não encontrado, ocorre uma situação similar para a *Thread*-2, porém as *threads* 3 e 5 exibem uma grande parcela de participação no intervalo de espera (15 segundos). Isso demonstra que essas duas *threads* continuam gerando eventos durante a execução pelo fato das auto-críticas não terem sido desabilitadas. O fato é confirmado também pelo número maior de eventos gerados por essas duas *threads* nas duas execuções de *"Auto-Crítica 2"*.

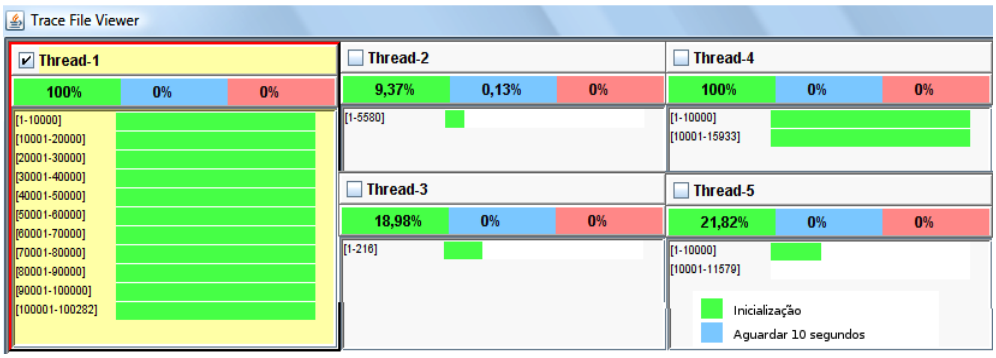


Figura 5.5: Rastro gerado para a *Execução 1* de *"Auto-Crítica 1"* destacando a inicialização e a espera dos 10 segundos após desabilitar as auto-críticas do *ArgoUML*.

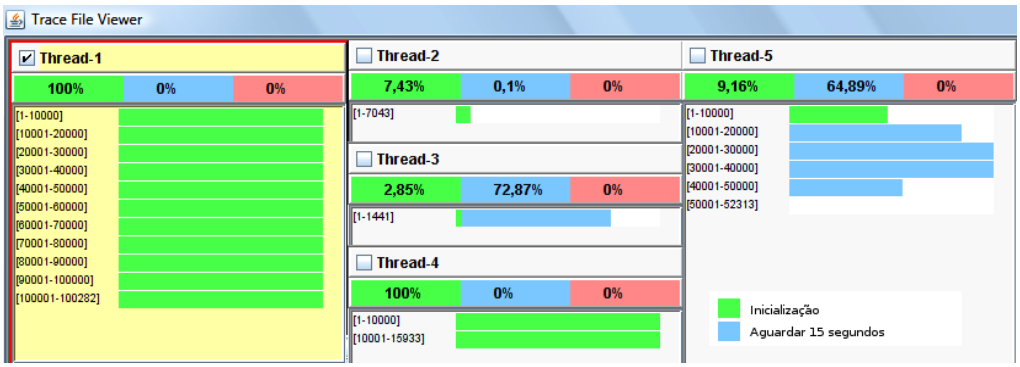


Figura 5.6: Rastro gerado para a *Execução 1* de *"Auto-Crítica 2"* destacando a inicialização e a espera dos 15 segundos com as auto-críticas do *ArgoUML* habilitadas.

3.3.2.3 Características de Interesse

Pela seleção de cada característica exercitada para a criação dos diagramas de classe (*Barra de Menu, Atalhos de Teclado, botão da Barra de Ferramentas*) pode ser observado no *TraceFileViewer* que apenas a *Thread-2* exibe eventos associados a essas características, conforme mostram as Figuras 5.7, 5.8 e 5.9. Estes correspondem a 30.9%, 27.42% e 29.53% dos eventos registrados para essa *thread,* os quais totalizam 87.85% do rastros. Os 12.15% restantes correspondem a eventos gerados antes, depois ou entre a demarcação da execução de cada característica.

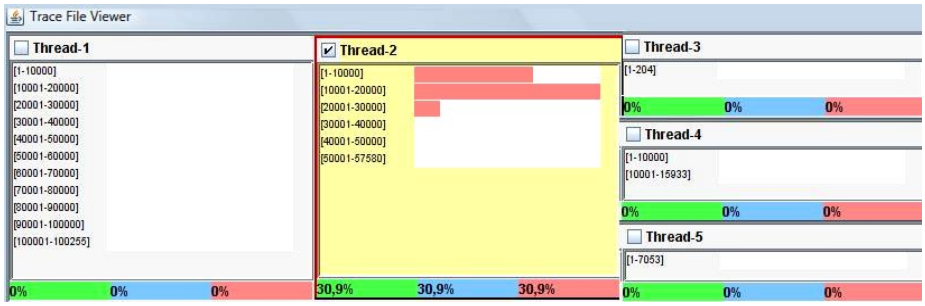


Figura 5.7: Participação característica *"Create Class Diagram by menu"* no rastro para *Execução 1* da *Sequência 1*.

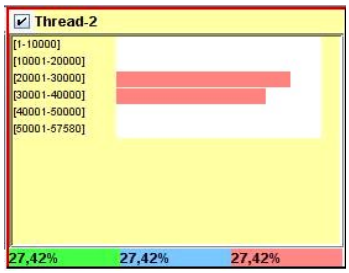


Figura 5.8: Participação da característica *"Create Class Diagram by shortcut"* no rastro para *Execução 1* da *Sequência 1* (*Thread-2*).

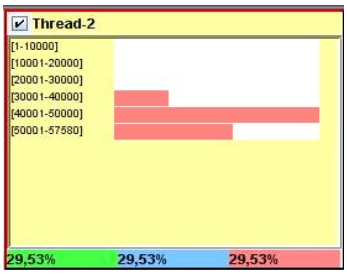


Figura 5.9: Participação da característica *"Create Class Diagram by toolbar"* no rastro para *Execução 1* da *Sequência 1* (*Thread-2*).

Assim, a análise da *Thread-2* seria suficiente para os propósitos da análise das características exercitadas.

3.3.3 Análise do Significado e Seleção de *Threads* de Interesse

As análises anteriores mostraram que os eventos efetivamente associados às características exercitadas foram gerados pela *Thread-2*. As *threads* 1 e 4 estão associadas a eventos de inicialização. As *threads* 3 e 5 estão associadas ao sistema de críticas do *ArgoUML.* As análises das *threads* 3 e 5 a seguir são feitas apenas a título de ilustração e para reforçar essas constatações.

3.3.3.1 Análise do rastro de threads com pequeno número de eventos

Entre as execuções das seqüências de 1 a 4, a *Thread-3* sempre mostra o menor número de eventos (Tabela 5.3, página 111). Por sua vez, as execuções 1 e 2 da *Seqüência 4* registram os valores mínimos (29 eventos cada). O maior número ocorre na *Execução 2* da Seqüência *3* (566 eventos). Embora pequenas quantidades de eventos no rastro não sejam a regra, nestes casos ainda é possível observar o rastro diretamente para verificar que tipos de eventos estão envolvidos. As tabelas 5.4 e 5.5Erro: Origem da referência não encontrado mostram as classes envolvidas nos rastros gerados pela T*hread*-3 na *Execução 1* da *Seqüência 4* (menor número de eventos) e na *Execução 2* da *Seqüência 3* (maior número de eventos), respectivamente.

Tabela 5.4. Classes presentes no rastro gerado pela *Thread-3* na *Execução da 1 Sequência 4*.

Pacote	***Classe***
`org.argouml.application`	`Main$1`

`org.argouml.cognitive`	`Designer`
	`ToDoList`
	`ToDoListEvent`
	`Translator`

`org.argouml.cognitive.ui`	`PriorityNode`
	`ToDoByPriority`
	`ToDoPane`

`org.argouml.i18n`	`Translator`

Embora a extração manual dessas informações seja possível, é um trabalho que toma tempo e sujeito a erros, pois seria necessário analisar cada linha do rastro.

Tabela 5.5.Classes presentes no rastro gerado pela *Thread-3* na *Execução 2* da *Sequência 3* (destacando classes e pacotes adicionais em relação à Tabela 5.5).

Pacote	***Classe***
`org.argouml.application`	`Main$1`

`org.argouml.cognitive`	`Designer`
	`ListSet`
	`ToDoList`
	`ToDoListEvent`
	`Translator`

`org.argouml.cognitive.ui`	`PriorityNode`
	`ToDoByPriority`
	`ToDoPane`

`org.argouml.i18n`	`Translator`
`org.argouml.kernel`	`Project` `ProjectManager`
`org.argouml.model`	`Model`
`org.argouml.model.uml`	`DefaultModelImplementation` `NSUMLModelFacade`
`org.argouml.ocl`	`CriticOclEvaluator`
`org.argouml.uml.cognitive`	`UMLToDoItem`

`org.argouml.uml.cognitive.critics`	`CrEmptyPackage`
	`WizMEName`
	`CrUnconventionalPackName`

`ru.novosoft.uml.model_management`	`MModelImpl`

A ferramenta permite a extração dessas informações diretamente. Como sugerido na seção anterior, a *Thread-3* está relacionada ao sistema de críticas do *ArgoUML*. A grande quantidade de classes pertencentes a pacotes contendo o termo *"cognitive"* ou com o prefixo *"Cr"* no nome reforça essa idéia, o que também pode ser confirmado na documentação do *ArgoUML* (ver *Cookbook-0.19.3*, página 53, Seção *"5.2 - Critics and other Cognitive Tools"* (TIGRIS.ORG, 2008)). Pode ser observado que a segunda listagem possui todas as classes da primeira e mais 12 adicionais (em destaque). Essa diferença pode ter ocorrido pelo intervalo de tempo decorrido até que as críticas fossem manualmente desabilitadas, o que inevitavelmente ocasionaria variações. As classes adicionais ou estão relacionadas diretamente ao sistema de críticas ou são classes de propósito geral, como *Project* e *Model*, que guardam as abstrações para o projeto e o repositório de modelos contidos nele. Isso continua sustentando a idéia de que essa *thread* esteja associada ao sistema de críticas.

Para as *threads* 2 e 5, é praticamente inviável uma análise manual das classes envolvidas no rastro. Essas *threads* possuem, respectivamente, uma média de 47.759 e 6.753 eventos entre as execuções realizadas. Avaliar as classes envolvidas em rastros com tantos eventos seria uma tarefa muito dispendiosa. Isto reforça a importância do uso de uma ferramenta para apoiar este tipo de atividade.

3.3.3.2 Participação dos pacotes no rastro e inferência da semântica

Os dois pacotes raízes presentes no rastro gerado para a *Execução 1* da Seqüência *1* são `org` e `ru`. Selecionando os dois pacotes na ferramenta é possível visualizar sua participação nos eventos do rastro como mostrado na Figura 5.10. Exceto pela *Thread-1*, a participação do pacote `ru` é mínima nas demais *threads*.

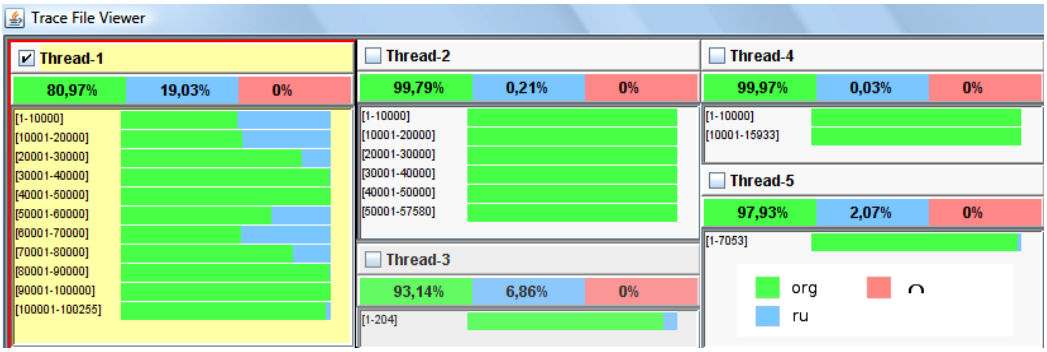


Figura 5.10: Participação dos pacotes `org` e `ru` no rastro da *Execução 1* da *Seqüência 1*.

Pela expansão dos pacotes `org` e `ru`, os pacotes principais no rastro são obtidos: `org.argouml`, `org.tigris.gef`, `org.tigris.toolbar`, `ru.novosoft.uml`. As participações desses pacotes são obtidas por uma análise similar à anterior e resumidas na Tabela 5.6.

De acordo com os dados da Tabela 5.6, `org.argouml` é o pacote predominante no rastro, o que de certa forma era esperado pois trata-se do pacote principal da aplicação. Os demais pacotes são *frameworks* utilizados, que por sua vez representam projetos à parte e compartilhados por outros projetos além do *ArgoUML*. As *threads* 1 e 2 possuem a presença mais expressiva dos *frameworks* (19,30% para `ru.novosoft.uml` e 24,13% para `org.tigris.gef`, respectivamente). O fato da *Thread-2* ter uma participação considerável do pacote

Tabela 5.6: Participação dos pacotes nas *threads* (*Execução 1,* Seqüência *1).*

	Thread-1	Thread-2	Thread-3	Thread-4	Thread-5
org.argouml	78,56%	73,11%	93,14%	97,46%	96,91%
org.tigris.gef	1,78%	24,13%	0,00%	2,51%	1,02%
org.tigris.toolbar	0,63%	2,56%	0,00%	0,00%	0,00%
ru.novosoft.uml	19,30%	0,21%	6,86%	0,03%	2,07%

`org.tigris.gef` sugere que essa *thread* possua mais eventos relacionados a interação com interface gráfica, já que o pacote `org.tigris.gef` está relacionado a esse tipo de funcionalidade. O mesmo pode ser inferido sobre o pacote `org.tigris.toolbar` que é o terceiro mais expressivo nessa *thread*.

Considerando a maior participação do pacote `org.argouml` no rastro e percorrendo seus subpacotes através do *TraceFileViewer*, pode-se verificar um dado interessante: o subpacote `org.argouml.cognitive` responde por 35,29% e 37,33% dos eventos no rastro das *threads* 3 e 5, respectivamente, enquanto o subpacote `org.argouml.uml.cognitive` respondem por 31,37% e 48,86%. Somando os eventos dos dois pacotes tem-se 66,66% e 86,19% do rastro das *threads* 3 e 5 cobertos. Como dito na Seção 4.5.1, os pacotes *"cognitive"* tem relação com o sistema de auto-críticas do ArgoUML. Isso fica ainda mais claro pela verificação de que `org.argouml.cognitive.critics` e `org.argouml.uml.cognitive.critics` são os subpacotes com maior participação dentre os subpacotes *"cognitive".* Isso confirma a relação das *threads* 3 e 5 com o sistema de auto-críticas do ArgoUML. É importante notar ainda que essas discrepâncias em pacotes *"cognitive"* não acontece nas demais *threads*. Na *Thread-2* por exemplo, os pacotes com maior representatividade são `org.argouml.model` (59,54%), seguido por `org.tigris.gef.base` (22,03%). Nessa *thread*, os pacotes `org.argouml.cognitive` e `org.argouml.uml.cognitive` respondem por apenas 0,6% e 0,03% do eventos.

Com isso, fica claro que as *threads* 1 e 3 estão realmente associadas ao sistema de auto-críticas do *ArgoUML*. A *Thread-2* está relacionada as características exercitadas, sendo que `org.argouml.model` e `org.tigris.gef.base` são os pacotes que contribuem com o maior número de eventos.

3.3.4 Análise de Elementos do Código relacionados às Características

Com base nas constatações anteriores sobre as *threads*, a partir dessa seção, a análise da relação entre elementos do código e características será focada no rastro da *Thread-2*, a qual mostrou ser a única a gerar eventos diretamente associados às características de interesse exercitadas.

3.3.4.1 Análise de Pacotes

Através da opção de listagem de pacotes exclusivos das características, nenhum pacote foi apontado como específico de uma delas. Isso mostra que ou os pacotes participam em mais de uma característica ou não participam em nenhuma delas.

A Figura 5.11 apresenta os gráficos obtidos pela expansão dos pacotes principais. As matrizes mostram, respectivamente, o número de elementos do código distintos (pacotes, classes e métodos) e eventos do rastro pertencentes aos pacotes principais e que participam nas características exercitadas. A coloração das células é praticamente a mesma em cada linha nas matrizes de Pacotes, Classes e Métodos. Nas matrizes de Eventos há uma pequena diferença perceptível apenas nas linhas 1 e 2 (pacotes `org.argouml` e `org.tigris.gef`). Isso indica que a participação em número de eventos do pacote `org.tigris.gef` (linha 2) é menor na característica *"Create Class Diagram by shortcut"* (coluna 2) quando comparada a participação desse mesmo pacote nas outras características (colunas 1 e 3). Isto é percebido apenas nas escalas *Gray*, *Hot Cold* e *Hot Cold Log*. O pacote com maior participação nas características em todos os casos é o pacote `org.argouml`. A participação dos demais pacotes sempre fica numa faixa inferior a 25% (ver escalas lineares *Discrete Levels*, *Gray* e *Hot Cold*). Apenas ressaltando, o valor percentual só pode ser inferido das matrizes baseado na coloração das células para as escalas lineares. As escalas logarítmicas transformam o valor das métricas, logo, as cores só podem ser utilizadas para situar a métrica na escala logarítmica e não para representar seu valor percentual real.

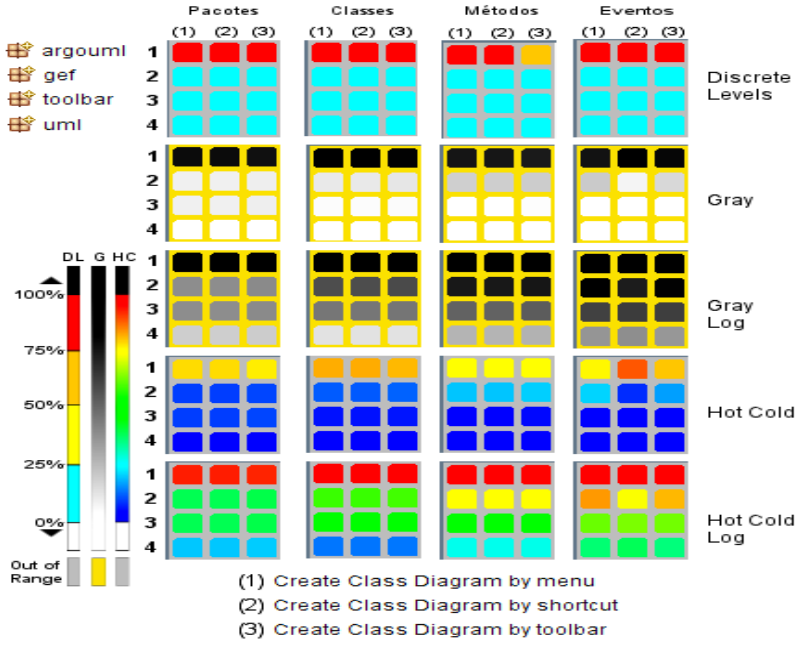


Figura 5.11: Matrizes de visualização mostrando o número distinto de elementos do código (pacotes, classes e métodos) e eventos derivados do rastro gerado pela *Thead-2* na *Execução 1* da *Sequência 1*, enfatizando os pacotes principais envolvidos no rastro.

As Tabela 5.7 a Tabela 5.14 mostram numericamente a participação dos pacotes principais, em termos absolutos e percentuais (sobre o total de elementos associados às características *menu*, *shortcut* e *toolbar*) confirmando as interpretações visuais. Até esse ponto, o que pode ser observado é a maior contribuição do pacote `org.argouml` na execução das características em todos os níveis (número de pacotes, classes, métodos e eventos), o que já havia sido constatado anteriormente.

A expansão de todos os pacotes pode fornecer uma visão geral do espalhamento das características no código. Porém, muitas informações podem ser irrelevantes ou redundantes. Assim, a Figura 5.12 é obtida excluindo pacotes que exibem informações redundantes ou irrelevantes para a análise, discutidas a seguir. Embora todos os pacotes contenham eventos no rastro, alguns não tem eventos associados às características de interesse tais como `org.argouml.language`, `org.argouml.moduleloader`, `org.argouml.ocl`, `org.argouml.pattern`. Há ainda pacotes relacionados ao sistema de autocríticas (contendo o termo "*cognitive*") e que não são de interesse para esta análise. Alguns pacotes, tais como `org` e `org.tigris` que compilam as informações de `org.argouml`, `org.tigris.gef` e `org.tigris.toolbar`, sumarizam informações que poderiam ser interessantes

Tabela 5.7: Pacotes compartilhados - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	26	26	24
org.tigris.gef	3	3	3
org.tigris.toolbar	3	3	3
ru.novosoft.uml	1	1	1
Total	33	33	31

Tabela 5.8: Pacotes compartilhados (%) - *Thread-2,* Seq-*1, Exec-1.*

	menu	shortcut	toolbar
org.argouml	78,79%	78,79%	77,42%
org.tigris.gef	9,09%	9,09%	9,68%
org.tigris.toolbar	9,09%	9,09%	9,68%
ru.novosoft.uml	3,03%	3,03%	3,23%

Tabela 5.9: Classes compartilhadas - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	145	146	136
org.tigris.gef	20	20	20
org.tigris.toolbar	10	10	10
ru.novosoft.uml	1	1	1
Total	176	177	167

Tabela 5.10: Classes compartilhadas (%) - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	82,39%	82,49%	81,44%
org.tigris.gef	11,36%	11,30%	11,98%
org.tigris.toolbar	5,68%	5,65%	5,99%
ru.novosoft.uml	0,57%	0,56%	0,60%

Tabela 5.11: Métodos compartilhados - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	503	501	473
org.tigris.gef	129	129	129
org.tigris.toolbar	29	29	32
ru.novosoft.uml	5	5	5
Total	666	664	639

Tabela 5.12: Métodos compartilhados (%) - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	75,53%	75,45%	74,02%
org.tigris.gef	19,37%	19,43%	20,19%
org.tigris.toolbar	4,35%	4,37%	5,01%
ru.novosoft.uml	0,75%	0,75%	0,78%

Tabela 5.13: Eventos compartilhados - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	13.637	14.045	13.683
org.tigris.gef	3.671	1.221	2.816
org.tigris.toolbar	451	478	470
ru.novosoft.uml	36	42	33
Total	17.795	15.786	17.002

Tabela 5.14: Eventos compartilhados (%) - *Thread-2,* Seq-*1, Exec-1*.

	menu	shortcut	toolbar
org.argouml	76,63%	88,97%	80,48%
org.tigris.gef	20,63%	7,73%	16,56%
org.tigris.toolbar	2,53%	3,03%	2,76%
ru.novosoft.uml	0,20%	0,27%	0,19%

para ver o impacto conjunto de seus sub pacotes numa característica. Porém como *GEF* e *Toolbar* são *frameworks* independentes do *ArgoUML*, em princípio, não faz muito sentido para esta análise ver o efeito de pacotes como `org.tigris` (que engloba *GEF* e *toolbar*) ou `org` (que engloba `org.argouml` e `org.tigris`). Outros pacotes trazem informação redundante, por exemplo, `ru` e `ru.novosoft` são os pacotes raízes de `ru.novosoft.uml` e não contém nenhum outro pacote direto (*"Nro.DirectPackages"* = 1, na ferramenta) ou elemento do código (*"Nro.DirectClasses" = 0,* na ferramenta) participante no rastro das características de

interesse, assim a informação visual exibida pelos três pacotes é a mesma na prática. Um exemplo similar ocorre entre `org.argouml.uml.ui.behavior` e `org.argouml.uml.ui.behavior.common_behaviour`. Um caso mais sutil ocorre quando um pacote contém contém elementos participantes no rastro ou mais de um pacote direto, porém, apenas um de seus subpacotes tem elementos que participam no rastro das características de interesse. Por exemplo, `org.argouml.uml.diagram.use_case` tem uma classe direta, porém apenas seu subpacote `org.argouml.uml.diagram.use_case.ui` tem elementos participantes no rastro das características de interesse. O mesmo acontece com `ru.novosoft.uml` cuja participação nas características se deve apenas ao seu subpacote `ru.novosoft.uml.model_management`.

Com um foco inicial nas matrizes *Hot Cold* da Figura 5.12, é visto que, embora haja um número razoável de pacotes para serem considerados (32), apenas alguns tem um destaque maior nessas matrizes. Para facilitar a leitura, a linha correspondente aos pacotes é colocada após sua citação entre parênteses. A matriz de Pacotes destaca o pacote `org.argouml (1)` e seu subpacote `org.argouml.uml (14)`, e de forma sutil `org.argouml.ui (9)`, `org.argouml.uml.diagram (15)` e `org.argouml.uml.ui (20)`. Possivelmente, esses são os pacotes com o maior número de módulos (implementados em seus subpacotes) atrelados às características de interesse. Passando para a matriz de Classes, alguns dos pacotes citados tem seu destaque reforçado (`org.argouml`, `org.argouml.ui`, `org.argouml.uml`, `org.argouml.uml.ui`), outros ofuscados (`org.argouml.uml.diagram`), e alguns ficam mais evidentes (`org.argouml.ui.explorer (11)` e seu subpacote `org.argouml.ui.explorer.rules (12)`). Pode ser inferido que as classes participantes no rastro das características estão bem balanceadas entre os pacotes, não havendo nenhum pacote com um número muito destoante de classes. Isso mostra que, neste caso, as duas matrizes são praticamente intercambiáveis por terem uma estrutura muito próxima. Similarmente, a matriz de Métodos, acompanha a estrutura das matrizes de Pacotes e Classes. A única diferença está no sutil destaque que aparece para os pacotes `org.argouml.model (5)` (e seu subpacote `org.argouml.model.uml (6)`) e `org.tigris.gef` (26) (e seu

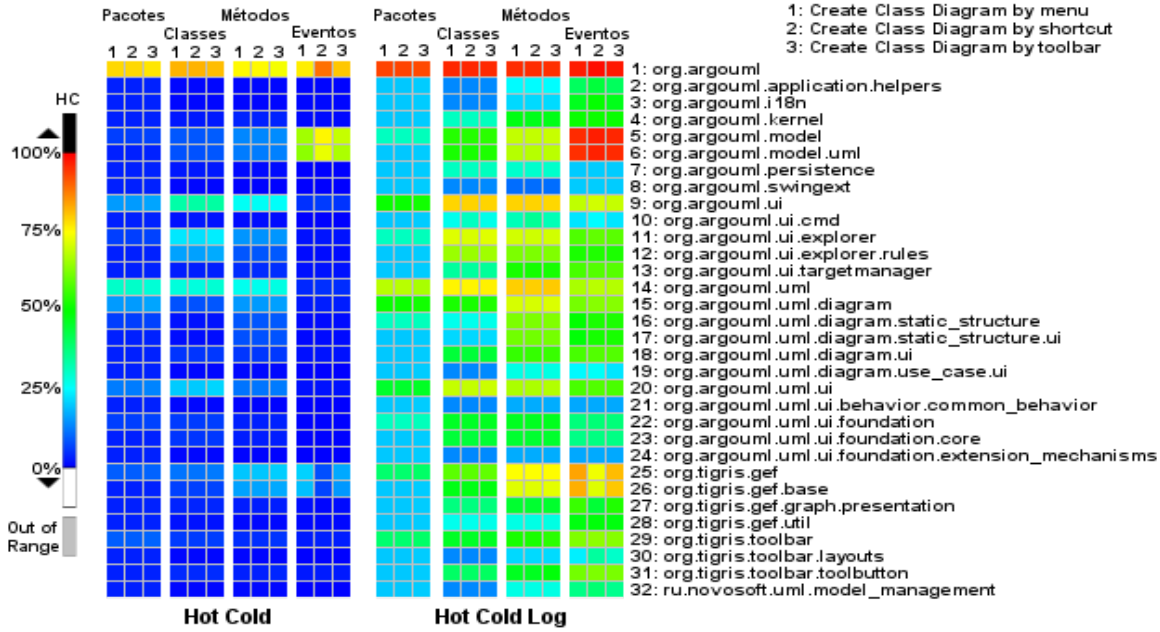


Figura 5.12: Matrizes de visualização mostrando o número distinto de elementos do código (pacotes, classes e métodos) e eventos derivados do rastro gerado pela *Thead-2* na *Execução 1* da Seqüência *1*, enfatizando apenas pacotes que contém elementos participantes nas características.

subpacote `org.tigris.gef.base (27)`), que passavam quase imperceptíveis frente aos demais nas matrizes de Pacotes e Classes. Em suma, as matrizes de Pacotes, Classes e Métodos mostraram estruturas equivalentes para a análise de pacotes (linhas 1 a 32) o que permite utilizar qualquer uma delas sem grandes perdas na interpretação para observar relação desses pacotes com as características (colunas 1 a 3). Elas mostram também que há um número relativamente pequeno de pacotes associados as características de interesse, o que pode ser um bom sinal para uma atividade de compreensão ou manutenção. Podem ser destacados os subpacotes de `org.argouml` evidenciados na matriz (`org.argouml.ui`, `org.argouml.uml`, `org.argouml.uml.ui` e `org.argouml.uml.diagram`) e `org.tigris.gef` (com seu subpacote `org.tigris.gef.base`).

Continuando a análise sobre as matrizes na escala *Hot Cold* da Figura 5.12, a matriz de Eventos é a que mais destoa das demais. Essa matriz reflete a quantidade de eventos no rastro relacionado às características de interesse e, portanto, é facilmente influenciada por eventos redundantes como laços, recursões e seqüências de chamadas recorrentes. Pode ser observado que `org.argouml (1)` continua tendo um destaque maior, porém os pacotes `org.argouml.model (5)` e

seu subpacote org.argouml.model.uml (6) ganham mais ênfase que os demais nessa matriz. Há um sutil destaque em org.tigris.gef (26) e seu subpacote org.tigris.base (27). Pode ser inferido com isso que os elementos em org.argouml responsáveis pela maior parte dos eventos do rastro estão em org.argouml.model, especialmente em seu subpacote org.argouml.model.uml. Essa poderia ser uma informação importante, por exemplo, para a análise de performance do sistema. Outro detalhe que pode ser observado é a diferença de tonalidade da coluna 2 nas linhas dos elementos com maior destaque (linhas 1, 5, 6, 25, 26). Enquanto as colunas 1 e 3 são similares nessas linhas, a coluna 2 indica que há uma concentração maior de eventos nas linhas 1, 5 e 6. Isso indica que org.argouml.model, org.argouml.model e org.argouml.model.uml (linhas 1, 5 e 6) são os pacotes com maior número de eventos associados à característica *"Create Class Diagram by shortcut"* (coluna 2), enquanto org.tigris.gef e org.tigris.base (linhas 25 e 26) tem participação equivalente aos demais pacotes.

As matrizes na escala *Hot Cold Log* são espelhos das *Hot Cold*, porém mostram as informações com um refinamento maior. Essas matrizes oferecem uma melhor distribuição dos elementos pela escala. Podem ser observadas três categorias de pacotes: os de maior destaque (laranja) contendo org.argouml (1); os de destaque intermediário (tons de verde) encabeçados por org.argouml.uml (14) e todos os pacotes citados com destaque sutil na matriz correspondente *Hot Cold*; e os de menor destaque (tons de azul) encabeçados por pacotes sem destaque na matriz de Pacotes *Hot Cold* tais como org.argouml.model (5) e org.argouml.ui.explorer (11). O mesmo acontece para as demais matrizes (Classes, Métodos e Eventos), as quais passam a ter uma nova caracterização sobre a participação dos pacotes nas características, que fornecem um melhor agrupamento dos pacotes segundo a ordem de grandeza de suas métricas. É importante destacar que as análises sobre as matrizes *Hot Cold* não são invalidadas com as matrizes *Hot Cold Log*.

As análises podem ser refinadas com o uso das matrizes *Hot Cold Log*, em especial na presença de métricas envolvendo valores de ordens muito distantes, o que é usual nas matrizes de Eventos. Nesses casos, elementos que eram ofuscados por elementos dominantes (tais como, org.argouml, org.argouml.model e

`org.argouml.model.uml` na matriz de Eventos) são revelados. Essas categorizações permitem manter uma noção de ordem a medida que os elementos são explorados pela expansão das matrizes, o que não seria possível quando um nível uniforme fosse alcançado. Como exemplo, suponha que o analista optasse pela busca de elementos responsáveis pelo maior número de eventos no rastro. Com base na matriz de Eventos *Hot Cold*, pela ordem natural seria examinado *org -> argouml -> model -> uml* e depois *org -> tigris -> gef base*. Feito isso e baseando-se apenas na matriz *Hot Cold*, não há critério algum que mostre quais os próximos pacotes a serem examinados. Porém, usando a matriz *Hot Cold Log*, após examinar os pacotes com maior destaque (`org.argouml.model.uml` e `org.tigris.gef.base`), poderia se caminhar pelos pacotes em tons de verde, azul claro e finalmente azul escuro.

3.3.4.2 Análise de Classes

A análise de pacotes está no nível mais alto de granularidade e fornece uma primeira triagem pelos elementos participantes no rastro. O caminho natural seria seguir para a análise das classes presentes nos pacotes e se necessário para os métodos. A Tabela 5.15 mostra as classes apontadas como exclusivas de cada características (ou seja, cujos eventos estejam associados apenas a uma delas) na *Execução 1 da Seqüência 1*. Apesar de haver classes exclusivas nessa execução, elas não parecem ter uma relação específica com essas características. Avaliando as outras execuções da *Seqüência 1*, essa informação se confirmou, pois houve variação nesses resultados. O mesmo aconteceu na análise das outras seqüências. Isso mostra que eventualidades durante a execução e a própria ordem de execução das características podem afetar os resultados, não sendo possível afirmar com certeza que uma característica é estritamente dependente de uma classe.

Tabela 5.15: Classes exclusivas das características exercitadas na Execução 1 da Seqüência 1 (Thread-2).

Características	*Classes Exclusivas*
menu	org.argouml.ui.explorer.ExplorerTree$ProjectPropertyChangeListener
shortcut	org.argouml.ui.ZoomSliderButton
	org.argouml.ui.cmd.ActionFind
toolbar	-

A Figura 5.13 mostra as classes contidas nos pacotes listados na Figura 5.12. Apenas classes com alguma participação nas características são mostradas. As matrizes de Métodos e Eventos foram usadas por serem as únicas a produzir alguma variação nas tonalidades das células ao serem consideradas classes (Capítulo 3 – *Featincode*, Seção 4.5.1, Tabela 3.1). A escala *Hot Cold Log* foi usada por agrupar melhor as classes segundo o grau de participação nas características. Como ocorre para pacotes, embora haja um número considerável de classes (167), apenas algumas tem uma participação mais expressiva. Dentre as classes de maior destaque na matriz de Eventos estão `EventKey` e `EventListenerHashMap` (linhas 10 e 11). Em geral, as classes com maior destaque na matriz de Métodos, também tem destaque considerável na matriz de Eventos. Como exemplo tem-se as classes `Project` (4), `NSUMLModelFacade` (19), `TargetManager` (86), `UMLClassDiagram` (91) e várias classes do pacote `org.tigris.gef.base` (linhas 137 a 147), tais como `Editor`, `Globals` e `LayerPerspectiveMutable`.

A Figura 5.14 mostra as 30 classes com maior destaque em cada matriz ordenadas de acordo com a coloração das células na escala *Hot Cold Log*. A diferença na ordem das classes entre as matrizes se deve ao uso de métricas distintas em cada uma delas (número de métodos distintos e número de eventos, respectivamente). Essas classes representam um bom ponto de partida para explorar o código fonte associado às características exercitadas. Cada matriz dá uma perspectiva diferente: a primeira mostra classes cujas API foram mais exploradas durante a execução (matriz de Métodos); a segunda mostra em quais classes os eventos gerados durante a execução mais se concentraram (matriz de Eventos). As classes nas primeiras posições da matriz de Métodos são classes chaves na execução do *ArgoUML* e, mais especificamente, na criação dos diagramas de classes, segundo a documentação no código fonte das classes. A `UMLClassDiagram` (1) é a representação para o diagrama de classes no *ArgoUML*, sendo responsável pelo acesso às principais funcionalidades desse tipo de diagrama. A classe `NSUMLModelFacade` (2) representa uma classe de fachada para o subsistema de Modelo do *ArgoUML*. A classe `Editor` (3) é a principal classe para a manipulação dos elementos gráficos do framework GEF. A classe `Project` (5) é a estrutura de dados que guarda o projeto corrente, sendo responsável por gerenciar a lista de diagramas e modelos UML criados. Por outro lado, na matriz de Eventos, a

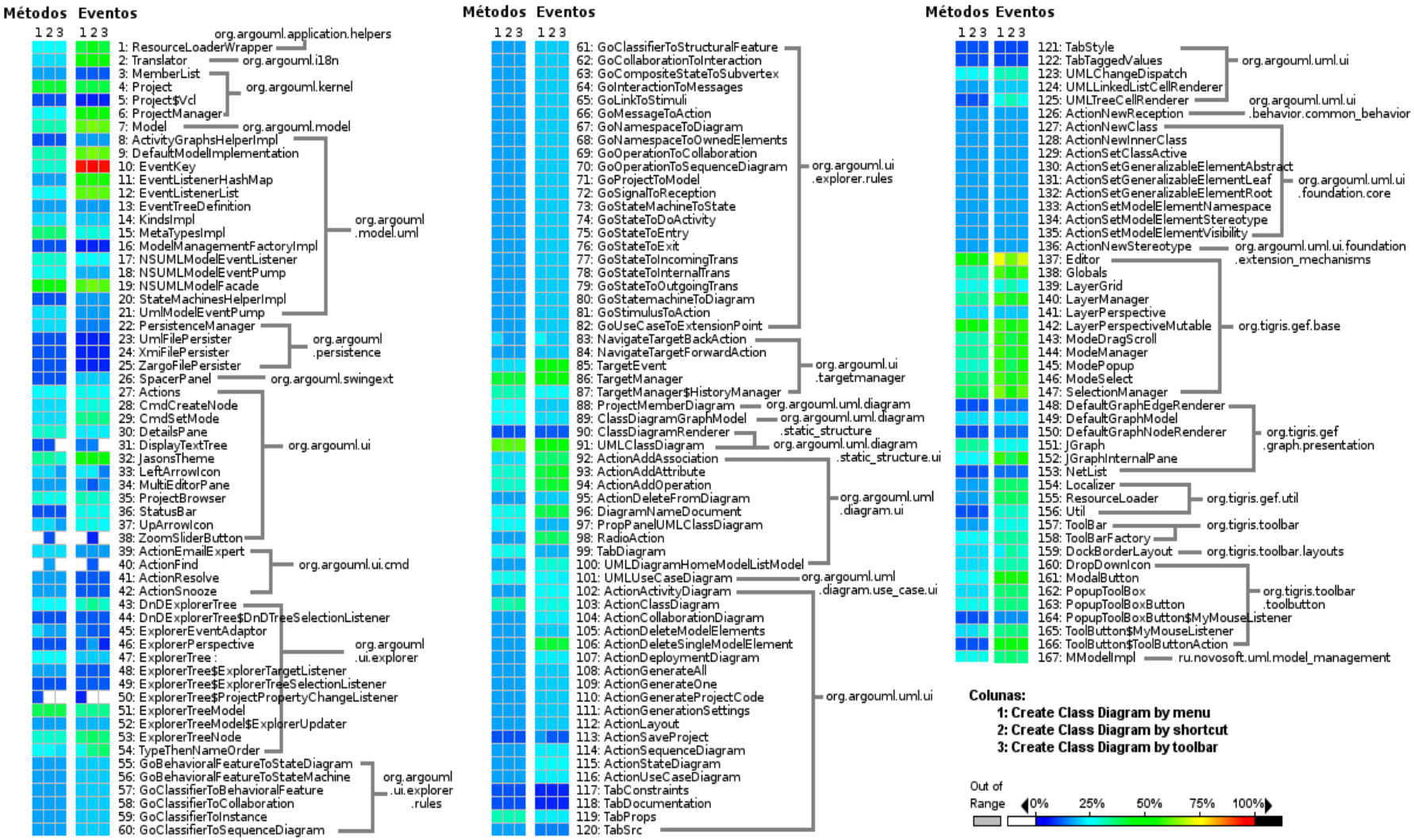


Figura 5.13: Matrizes de visualização mostrando o número distinto de métodos e eventos derivados do rastro gerado pela *Thead-2* na *Execução 1* da Seqüência *1*, enfatizando apenas classes participantes nas características (Escala *Hot Cold Log*).

classe `EventKey`, embora tenha o maior destaque, representa uma classe auxiliar presente no código fonte `UMLModelEventPump`, sendo utilizada em tabelas *hash* para o mapeamento de entre eventos e ouvintes no ArgoUML, fato que justifica a sua presença massiva no rastro.

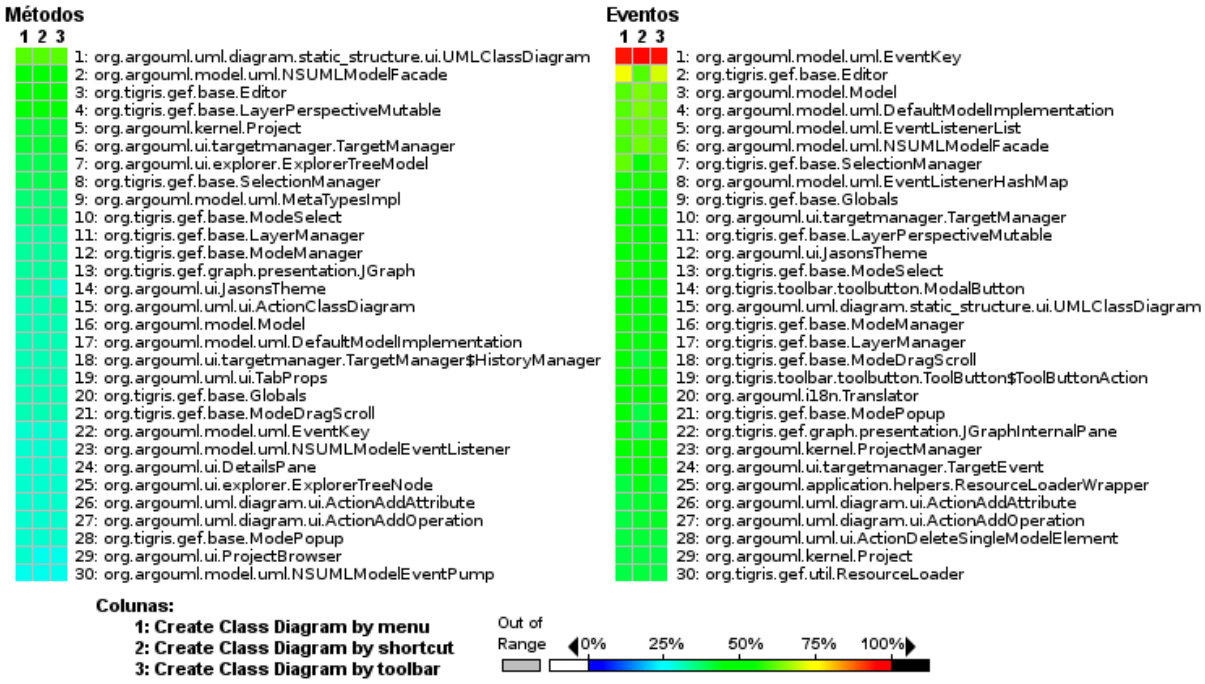


Figura 5.14: Classes participantes nas características ordenadas segundo a coloração das células e na escala *Hot Cold Log*.

3.3.4.3 Análise de Métodos

Neste ponto, a análise de métodos pode então ser realizada. A Tabela 5.16 mostra os métodos exclusivos de cada característica. As mesmas observações feitas sobre as classes no início da sessão anterior são válidas para os resultados com métodos. Porém, dentre estes métodos, os métodos da classe `ToolButton` se mantiveram em todas execuções e, de fato, parecem ser os mais representativos para a criação de diagramas de classe por meio dos botões na barra de ferramentas (*Toolbar*). Dado que as características são muito similares, a presença desses métodos é a principal distinção apontada além, claro, da quantidade de eventos gerados no rastro.

Tabela 5.16: Métodos exclusivos das características

Características	*Métodos Exclusivos*
menu	org.argouml.kernel.ProjectManager.firePropertyChanged
	org.argouml.ui.ProjectBrowser.propertyChange
	org.argouml.ui.explorer.ExplorerEventAdaptor.propertyChange
	org.argouml.ui.explorer.ExplorerTree$ProjectPropertyChangeListener.propertyChange
	org.argouml.ui.targetmanager.NavigateTargetBackAction.isEnabled
shortcut	org.argouml.ui.ZoomSliderButton.getRealAction
	org.argouml.ui.cmd.ActionFind.isEnabled
	org.argouml.ui.cmd.ActionFind.shouldBeEnabled
toolbar	org.argouml.ui.JasonsTheme.getSystemTextFont
	org.argouml.uml.ui.ActionClassDiagram.isEnabled
	org.tigris.toolbar.toolbutton.ToolButton$MyMouseListener.mouseClicked
	org.tigris.toolbar.toolbutton.ToolButton$MyMouseListener.mousePressed
	org.tigris.toolbar.toolbutton.ToolButton$MyMouseListener.mouseReleased

A análise visual das matrizes também pode ser realizada. Para ilustrar, a Figura 5.15 mostra os métodos das classes `UMLClassDiagram` e `Project` participantes na execução das características. Embora não sejam os métodos que participem no maior número de eventos, os métodos `addDiagram`, `addDiagramMember` e `addMember` merecem destaque na classe `Project` em função de sua relação com a adição de diagramas e elementos do modelo ao projeto. Já na classe `UMLClassDiagram`, a presença do construtor (`<init>`) na execução das características é mais uma evidência da relação dessa classe com a criação dos diagramas de classe.

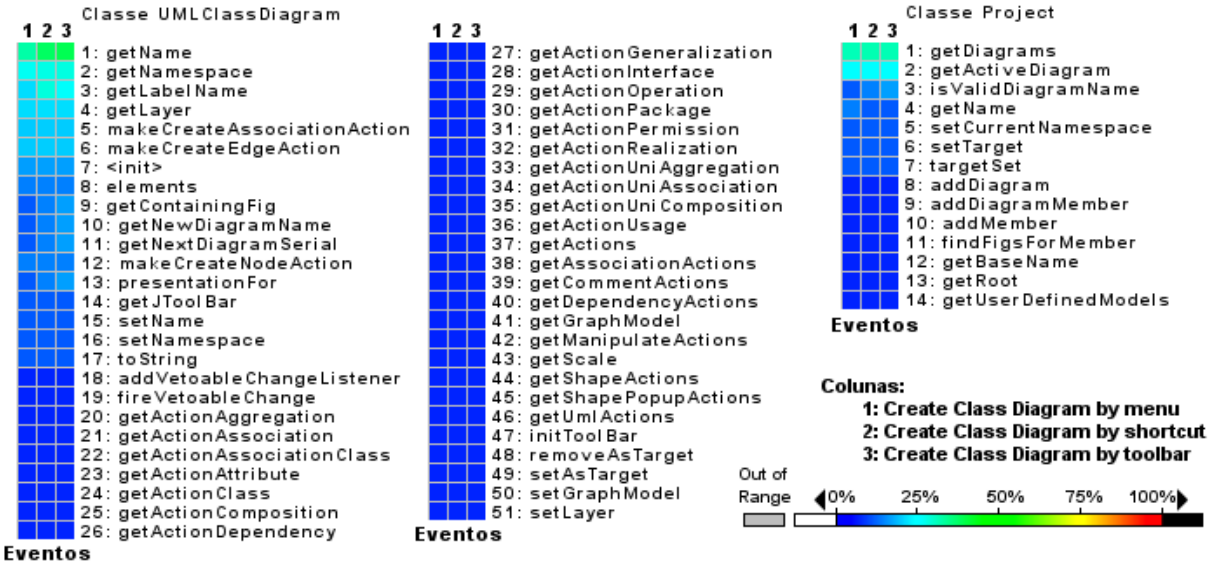


Figura 5.15: Métodos das classes UMLClassDiagram e Project participantes nas características.

3.3.5 Análise de Similaridade entre Elementos do Código

Exceto pelos elementos do código que participam apenas na inicialização, praticamente todos os demais elementos tem alguma participação nas três características exercitadas. No nível de classes apenas `DisplayTextTree`, `ZoomSliderButton`, `ActionFind` e `ExplorerTree$ProjectPropertyChangeListener` deixam de participar em alguma das características (ver Figura 5.13, linhas 31, 38, 40 e 50, respectivamente). Assim, a análise de similaridade entre pacotes e entre classes através das matrizes é desnecessária para este caso, já que todas as características são comuns entre esses elementos.

3.3.6 Análise de Similaridade de Implementação entre Características

As mesmas colocações para a análise de similaridade entre elementos do código, são válidas para a análise de similaridade entre as características, pois essas características apresentam distinção no compartilhamentos de apenas quatro classes.

3.3.7 Redução do escopo para análise e compreensão do sistema

A Tabela 5.17 mostra métricas extraídas do rastro para os pacotes `org.argouml`, `org.tigris.gef`, `org.tigris.toolbar`, `ru.novosoft.uml` referentes a todas as *threads* executadas. A Tabela 5.18 mostra o percentual de elementos presentes no rastro sobre o total de elementos do pacote correspondente no código fonte (ver Tabela 5.1 para métricas do código fonte). Por exemplo, a linha 1 indica que 46,52% dos métodos de `org.argouml` estão presentes no rastro. A Tabela 5.19 mostra o percentual de elementos no rastro considerando todos os elementos do código fonte.

Tabela 5.17: Métricas extraídas do rastro (*Execução 1, Seqüência 1*).

	NOP	NOC+NOI	NOM
org.argouml	48	502	3860
org.tigris.gef	6	41	215
org.tigris.toolbar	3	10	34
ru.novosoft.uml	5	11	127
Total	62	564	4236

Tabela 5.18: Cobertura dos pacotes no código fonte pelos elementos presentes no rastro (*Execução 1, Seqüência 1*).

	NOP[1]	NOC+NOI[1]	NOM[1]
org.argouml	67,61%	38,76%	46,52%
org.tigris.gef	31,58%	13,49%	7,33%
org.tigris.toolbar	100,00%	31,25%	15,38%
ru.novosoft.uml	33,33%	4,42%	3,45%

Tabela 5.19: Projeção geral dos elementos do rastro sobre o total de elementos do código (*Execução 1, Sequência 1*).

	NOP[2]	NOC+NOI[2]	NOM[2]
org.argouml	44,44%	26,70%	25,52%
org.tigris.gef	5,56%	2,18%	1,42%
org.tigris.toolbar	2,78%	0,53%	0,22%
ru.novosoft.uml	4,63%	0,59%	0,84%
Total	57,41%	30,00%	28,00%

De acordo com as tabelas 5.18 e 5.19, a quantidade de elementos do código presentes no rastro de execução é bem inferior a quantidade de elementos total no código fonte do *ArgoUML* e das respectivas bibliotecas. Isso por si só já seria uma informação importante, atestando a intuição de que uma análise completa do código fonte seria desnecessária considerando-se o interesse em apenas parte do sistema.

As tabelas 5.21 a 5.22 mostram as mesmas métricas das tabelas anteriores, porém considerando apenas o rastro da *Thread-2*. Com base nesses resultados, a quantidade de elementos efetivamente associados às características é muito inferior tanto ao total de elementos presentes no código fonte, quanto aos elementos presentes no rastro e sem associação direta às características exercitadas. Isso demonstra que, uma vez que os elementos destacados pela ferramenta sejam relevantes para o entendimento do sistema com base nessas características, o esforço dispendido para o entendimento pode ser expressivamente reduzido. Tomando o exemplo das classes e interfaces, partiu-se de um universo de 2.185 elementos para 188 efetivamente associados as características, o que corresponde a 8,65% do total. Considerando apenas o pacote `org.argouml`, que corresponde à

Tabela 5.21: Métricas extraídas do rastro gerado pela *Thread-2* na *Execução 1* da *Seqüência 1*.

	NOP	NOC+NOI	NOM
org.argouml	26	147	508
org.tigris.gef	3	30	129
org.tigris.toolbar	3	10	32
ru.novosoft.uml	1	1	5
Total	33	188	674

Tabela 5.20: Cobertura dos pacotes no código fonte pelos elementos presentes no rastro gerados pela *Thread-2* na *Execução 1* da Seqüência *1* e associados às características exercitadas.

	NOP[1]	NOC+NOI[1]	NOM[1]
org.argouml	36,62%	11,35%	6,12%
org.tigris.gef	15,79%	9,87%	4,40%
org.tigris.toolbar	100,00%	31,25%	14,48%
ru.novosoft.uml	6,67%	0,40%	0,14%

Tabela 5.22: Projeção dos elementos do rastro gerados pela *Thread-2* na *Execução 1* da Seqüência *1* e associados às características exercitadas sobre o total de elementos do código.

	NOP[2]	NOC+NOI[2]	NOM[2]
org.argouml	24,07%	7,82%	3,36%
org.tigris.gef	2,78%	1,60%	0,85%
org.tigris.toolbar	2,78%	0,53%	0,21%
ru.novosoft.uml	0,93%	0,05%	0,03%
Total	30,56%	10,00%	4,46%

implementação efetiva do *ArgoUML*, partiu-se de um total de 1.295 classes e interfaces para 147, que correspondem a 11,35% dos elementos presentes no código fonte desse pacote e 6,73% do total considerando também o código fonte das demais bibliotecas envolvidas.

As Figuras 5.16 a 5.18 mostram a redução na quantidade de elementos a serem analisados. É mostrada a redução em termos de pacotes, classes e métodos respectivamente. Sem o conhecimento de quais elementos seriam mais relevantes para a análise, o universo seria composto por todos os elementos do código fonte. Pela execução do sistema, o universo seria reduzido aos elementos manifestados no rastro. Nem todos os elementos no rastro estão efetivamente relacionados às características de interesse, assim o escopo para a análise seria ainda mais reduzido pela seleção dos elementos com alguma relação com as características exercitadas. Por fim, constata-se que os eventos gerados por algumas threads,

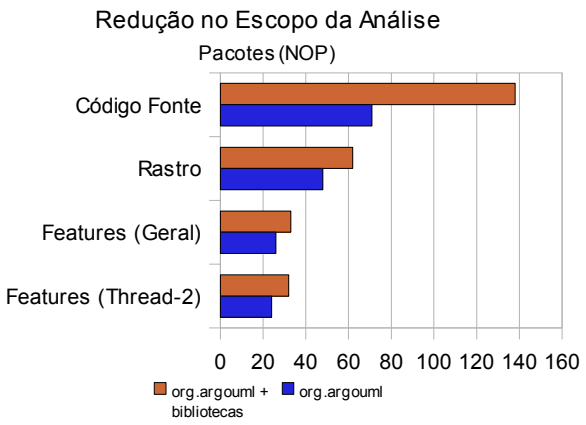


Figura 5.16: Redução no escopo da análise (pacotes) para Execução 1 da Seqüência 1.

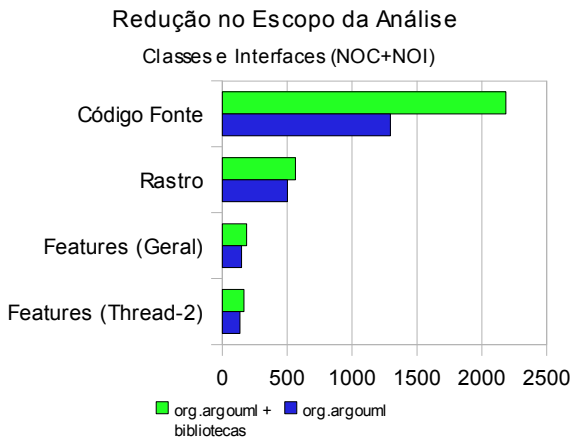


Figura 5.17: Redução no escopo da análise (classes e interfaces) para a Execução 1 da Seqüência 1.

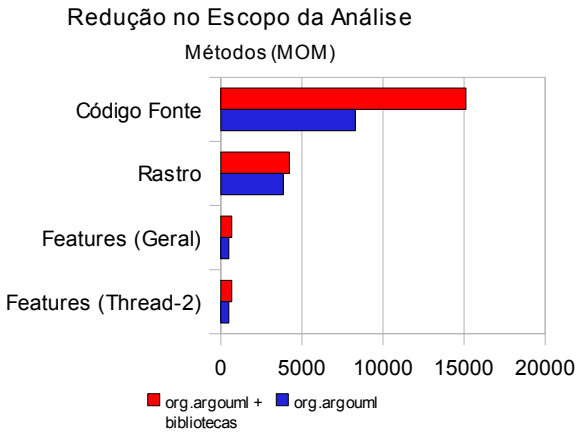


Figura 5.18: Redução no escopo da análise (métodos) para a Execução 1 da Seqüência 1.

podem não ser relevantes para a análise, como foi o caso das threads ligadas à inicialização (*Thread-1* e *4*) e geração de críticas automáticas (*Thread-3* e *5*). Como os eventos associados às características se limitaram apenas à *Thread-2*, o número de eventos associados a todas as *threads* (*Features – Geral*) e apenas para a *Thread-2* é o mesmo.

3.4 Discussão

Com este estudo é possível perceber a notória redução no número de

elementos a serem analisados com a finalidade de compreensão ou manutenção de um sistema pelo uso da metodologia e ferramenta proposta. A noção intuitiva de que uma análise exaustiva do código fonte é desnecessária quando o interesse está sobre um subconjunto das características presentes no sistema pôde ser confirmada. Foi verificado que muitos dos elementos previamente carregados pelo *ArgoUML* não são novamente acessados nas funcionalidades exercitadas. A análise visual das matrizes apontou elementos relevantes para as características exercitadas. Mesmo considerando a alta similaridade entre as características, foi possível classificar os elementos conforme sua participação no rastro. Esta similaridade ficou clara também pela alta semelhança na coloração das células correspondentes a cada características. Além disso, foi mostrada a influência de funcionalidades que permanecem em execução durante o exercício das características (neste caso, o sistema de auto-crítica do *ArgoUML*) e que se não forem interrompidas podem influenciar bastante na análise e interpretação dos resultados. Isso mostra a importância de se separar eventos gerados por diferentes *threads* e em reconhecer o significado dessas *threads*, confirmando a relevância do esquema de visualização proposto (especialmente, o *TraceFileViewer*).

Um dos pontos fracos da abordagem foi descoberto nas inconsistências entre classes e métodos tidos como exclusivas entre as características. Houve variação de parte desses elementos entre execuções diferentes, tanto pra mesma seqüência quanto para seqüências distintas. Porém, esta variação acontece em geral para elementos do código com pequena participação no rastro, fato que não invalida a relação entre características e elementos do código com grande participação.

4 Contexto 2: Características distintas com pontos em comum

Seção 4.1 traz os objetivos específicos para o contexto 2. Seção 4.2 descreve os roteiros de execução e na Seção 4.3 são mostradas as análises feitas e os resultados obtidos. As discussões sobre os resultados são apresentadas na Seção 4.4.

4.1 Objetivos específicos

O segundo contexto enfoca a análise de características distintas, porém com pontos em comum. Como no contexto anterior, pretende-se destacar elementos comuns e distintos entre roteiros de execução similares, além de inferir a semântica de eventos e *threads* do rastro. Adicionalmente, análises sobre a similaridade entre características, a similaridade entre elementos do código fonte e a influência da ordem de execução das características serão realizadas. Novamente, será mostrada a redução na quantidade de elementos a serem analisados com base nos elementos associados às características de interesse exercitadas.

4.2 Descrição dos Roteiros de Execução

A criação de diagramas é uma das características fundamentais e essenciais do *ArgoUML*. Na versão *0.19.3*, Quatro tipos de diagramas estão disponíveis: *Diagrama de Classes*, *Diagrama de Casos de Uso*, *Diagrama de Atividades* e *Diagrama de Implantação*. Entre esses diagramas há muitas características em comum e também várias características intrínsecas a cada um deles. A própria interface do *ArgoUML* apresenta alguns padrões de interação gerais para a criação e edição dos diagramas. Por exemplo, todos os diagramas criados são listados na área da interface denominada *"Explorer Pane"*, bem como os elementos envolvidos no modelo. Para editar um dos diagramas, é necessário selecioná-lo no *"Explorer Pane"*. A seguir, a área denominada *"Editor Pane"* é atualizada de acordo com o diagrama selecionado. Na parte superior do editor encontram-se botões de ferramentas que permitem inserir diferentes tipos de elementos nos diagramas. Cada diagrama possui um conjunto específico de ferramentas. Algumas delas são comuns entre diagramas distintos. Além disso, outra área da interface denominada *"Details Pane"* também é atualizada com informações específicas do diagrama corrente ou dos elementos selecionados nesse diagrama. Diante disto, os roteiros consistem na criação e edição desses quatro tipos de diagramas A Tabela 5.23 mostra alguns dos elementos comuns entre os diagramas disponíveis na versão *0.19.3* do *ArgoUML*. As linhas sombreadas mostram os elementos cobertos pelos roteiros de execução.

Tabela 5.23: Elementos comuns entre diagramas UML.

Elementos do Modelo		Diagrama			
		Classes	Casos de Uso	Atividades	Implantação
Ator			x		
Caso de Uso			x		
Relação	Associação	x	x		x
	Link				x
	Link de Comentário	x	x	x	x
	Generalização	x	x		x
	Extensão		x		
	Inclusão		x		
	Dependência	x	x		x
	Realização	x			x
Classe		x			
Classe de Associação		x			
Estado	(Inicial/Ação)			x	
Transição				x	
Pacote		x			
Interface		x			
Nó					x
Componente					x
Objeto					x
Comentário		x	x	x	x

Três roteiros de execução foram definidos. O *Roteiro de Execução 1* exercita as características de interesse através dos passos na *Sequência 1*. O *Roteiro de Execução 2* utiliza os passos na *Sequência 2 e* contém os mesmos passos da *Sequência 1*, porém numa ordem diferente. O *Roteiro de Execução 3* envolve a simples inicialização do *ArgoUML*. Antes de cada um dos roteiros, os passos definidos na *Sequência Prévia* foram realizados com o objetivo de reduzir o número de elementos não relacionados às características presentes no rastro.

4.2.1 Seqüência Prévia

1. Com o intuito de reduzir a quantidade de elementos carregados pelo ArgoUML e não necessariamente relacionados às características exercitadas, as opções de linha de comando a seguir foram usadas para a execução:
   1.1. *"-nosplash"*: impede a exibição do "splash" e do progresso do carregamento.
   1.2. *"-nopreload"*: impede o carregamento prévio de elementos do código.
   1.3. *"-norecentfile"*: impede o carregamento do projeto salvo após a última sessão de uso.

2. Desabilitar as funcionalidade de críticas automáticas, como feito no Contexto 1.
3. Carregar um projeto previamente salvo contendo os quatro diagramas que serão editados (Diagrama de Classes, Diagrama de Casos de Uso, Diagrama de Atividades, Diagrama de Implantação). Expandir o elemento correspondente ao Diagrama de Atividades para que seja necessário apenas selecioná-lo para iniciar sua edição durante a execução.

4.2.2 Sequência 1:

1. Draw Class Diagram (CD)
   1.1. Select CD on Explorer Pane
   1.2. Create Class
      a) Select Class Tool
      b) Insert Class in Editor Pane
   1.3. Create Interface
      a) Select Interface Tool
      b) Insert Interface in Editor Pane
   1.4. Create Relation (CD)
      a) Select Dependency Relation Tool (CD)
      b) Insert Dependency Relation in Editor Pane (CD)
   1.5. Create Comment (CD)
      a) Insert Comment clicking on Comment Tool (CD)
      b) Select Comment Link Tool (CD)
      c) Insert Comment Link between Class and Comment
2. Draw Use Case Diagram (UCD)
   2.1. Select UCD on Explorer Pane
   2.2. Create Actor
      a) Select Actor Tool
      b) Insert Actor in Editor Pane
   2.3. Create Use Case
      a) Select Use Case Tool
      b) Insert Use Case in Editor Pane

2.4. Create Relation (UCD)

a) Select Dependency Relation Tool (UCD)

b) Insert Dependency Relation in Editor Pane (UCD)

2.5. Create Comment (UCD)

a) Insert Comment clicking on Comment Tool (UCD)

b) Select Comment Link Tool (UCD)

c) Insert Comment Link between Actor and Comment

3. Draw Activity Diagram (AD)

3.1. Select AD on Explorer Pane

3.2. Create Initial State

a) Select Initial State Tool

b) Insert Initial State

3.3. Create Action State

a) Select Action State Tool

b) Insert Action State

3.4. Create Transition

a) Select Transition Tool

b) Insert Transition State

3.5. Create Comment (AD)

a) Insert Comment clicking on Comment Tool (AD)

b) Select Comment Link Tool (AD)

c) Insert Comment Link between Initial State and Comment

4. Draw Deployment Diagram (DD)

4.1. Select DD on Explorer Pane

4.2. Create Component

a) Select Component Tool

b) Insert Component

4.3. Create Node

a) Select Node Tool

b) Insert Node

4.4. Create Relation (DD)

a) Select Dependency Relation Tool (DD)

b) Insert Dependency Relation in Editor Pane (DD)

4.5. Create Comment (DD)

a) Insert Comment clicking on Comment Tool (DD)
b) Select Comment Link Tool (DD)
c) Insert Comment Link between Component and Comment

4.2.3 Sequência 2:

A seqüência 1 é repetida porém variando a ordem dos passos de execução como descrito a seguir: 4 (4.1, 4.5 a, 4.3, 4.2, 4.4, 4.5 b-c), 2 (2.1, 2.5 a, 2.3, 2.2, 2.4, 2.5 b-c), 3 (3.1, 3.5 a, 3.3, 3.5 b-c, 3.2, 3.4),1 (1.1, 1.3, 1.5 a, 1.2, 1.5 b-c, 1.4).

4.2.4 Inicialização:

Iniciar o *ArgoUML* aceitando a opção de marcação da inicialização. Após o carregamento e a apresentação da janela principal do sistema, encerrar a marcação da inicialização e sair do programa.

4.3 Análises e Resultados

Esta seção mostra as análises e resultados do estudo feito com o Contexto 2. Seção 4.3.1 resume os dados coletados em número de eventos por *thread* e para cada execução. Seção 4.3.2 mostra a análise do espalhamento das características pelo código. Seção 4.3.3 mostra a análise do significado e seleção das *threads*. Seção 4.3.4 analisa os elementos do código relacionados às características. A seções 4.3.5 mostram as análises de similaridade entre características. Seção 4.3.6 apresenta a redução de escopo para a análise. Seção 4.3.7 mostra a influência da ordem de execução das características sobre as interpretações.

4.3.1 Dados Coletados

A Tabela 5.24 resume a quantidade de eventos envolvida nas execuções dos roteiros 1, 2 e 3 do Contexto 2, para cada uma das *threads* disparadas. Pode ser observada certa diferença no número de *threads* iniciadas. Porém, dada a quantidade de eventos envolvidos nessas *threads* adicionais (*threads* 6 a 10), espera-se que estas threads não sejam tão significantes quando comparados às

Tabela 5.24: Número de eventos gerados pelos roteiros de execução do Contexto 2.

	Thread-1	*Thread-2*	*Thread-3*	*Thread-4*	*Thread-5*	*Thread-6*	*Thread-7*	*Thread-8*	*Thread-9*	*Thread-10*
Roteiro 1 (Sequência 1)										
Execução 1	100.214	1.996.775	29	15.932	2.527	4	4	58	..	..
Execução 2	100.214	2.082.535	29	15.932	2.527	1	4	4	58	..
Execução 3	100.214	2.122.977	41	15.932	2.527	1	1	4	4	58
Roteiro 2 (Sequência 2)										
Execução 1	100.156	2.164.372	41	15.932	2.527	1	4	4	58	..
Execução 2	100.214	2.067.599	29	15.932	1	4	4	58	..	..
Execução 3	100.214	2.074.473	41	15.932	2.527	4	4	58	..	..
Roteiro 3 (Inicialização)										
Execução 1	100.214	544	77	15.932	4.246	..	..	..	..	..
Execução 2	100.214	3.797	228	15.932	8.772	..	..	..	..	..

demais que chegam a ter milhões de eventos, como a *Thread-2*.

4.3.2 Análise do Espalhamento das Características pelo Rastro

Na Execução 1 do Roteiro 1, oito *threads* foram iniciadas. A Figura 5.19 mostra a participação de todas as características no rastro. Novamente fica clara a relação das características exclusivamente com os eventos da *Thread-2.*

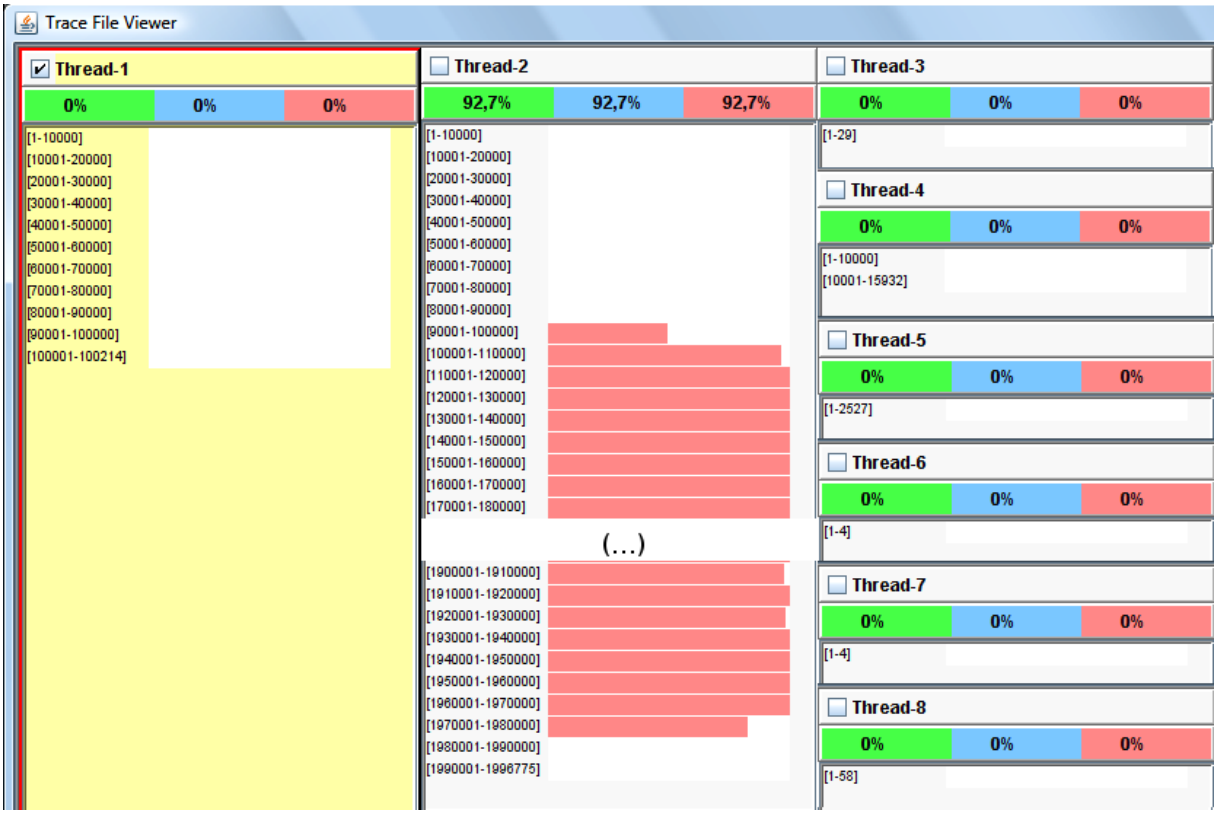


Figura 5.19: Participação das características no rastro da Execução 1 do Roteiro 2.

4.3.3 Análise do Significado e Seleção de *Threads* de Interesse

Como uma situação similar ocorrida no Contexto 1 se repete neste contexto, será feito aqui um breve comparativo entre os resultados.

Primeiramente, três novas *threads* surgiram. As *threads* 6 e 7 puderam ser analisadas diretamente pela pequena quantidade de eventos (4 eventos cada). A única chamada nessas duas *threads* foi para o método `accept` da classe `org.argouml.persistence.ZargoFilePersister`. A thread 8 foi similarmente examinada, sendo encontradas as chamadas `accept` e `getExtension`, da classe `org.argouml.persistence.ZargoFilePersister`, e `getExtension`, da classe `org.argouml.persistence.AbstractFilePersister`.

Como essas *threads* não apareceram no Contexto 1 e não estão associadas às características exercitadas neste contexto (Figura 5.19), possivelmente surgiram em função do carregamento do projeto com os diagramas de classe previamente criados, necessário para a execução do Roteiro 1 e 2. Isso é percebido também pelos eventos ocorridos antes do início da execução das características na *Thread-2* (cerca de 90 mil eventos antes das barras vermelhas indicando a participação das características no rastro). Para a comprovação dessas afirmações, seria necessário a realização de um novo experimento, o que por estar fora do escopo desse estudo não foi realizado. Pelo próprio contexto (carregamento do projeto e classes envolvidas), acredita-se que a explicação para o surgimento dessas *threads* seja razoável.

Quanto às demais *threads*, as mesmas observações feitas para o Contexto 1 são válidas para o Contexto 2. As *Threads* 1 e 4 continuam associadas à inicialização e as *Threads* 3 e 5 ao sistema de críticas. Assim, a *Thread-2* é novamente a única *thread* de interesse para a análise.

4.3.4 Análise de Elementos do Código relacionados às Características

As subseções a seguir mostram as análises de pacotes (Seção 4.3.4.1), classes (Seção 4.3.4.2) e métodos (Seção 4.3.3) relacionados às características.

4.3.4.1 Análise de Pacotes

O único pacote apontado como exclusivo para a característica *"Draw Class Diagram"* considerando todas as execuções dos roteiros 1 e 2 é `org.argouml.language.helpers`. Similarmente, para a característica *"Draw Activity Diagram"* apenas o pacote `activity_graphs` em `org.argouml.uml.ui.behavior`, foi apontado como exclusivo também. Para as demais características, nenhum pacote pode ser considerado exclusivo. O critério adotado para a exclusividade foi o de que o pacote tenha sido apontado pela ferramenta como exclusivo para a mesma característica em todas as execuções dos roteiros 1 e 2. Essa medida foi adotada para amenizar a influência da ordem de execução na interpretação dos resultados.

A Figura 5.20 exibe a participação dos pacotes nas características para a Execução 1 do Roteiro 1. Os pacotes estão ordenados de acordo com a coloração das células, que refletem o número de métodos distintos e o número de eventos relacionados às características, respectivamente. Diferentemente do Contexto 1, a matriz mostra que há variação na participação dos pacotes entre as características. Além disso, é possível ver alguns pacotes que tem uma concentração maior em algumas características, embora participem de todas, como por exemplo `static_structure` em `org.argouml.uml.diagram.static_structure`.

A Tabela 5.25 resume os pacotes com maior concentração individual nas características (entre parênteses estão os números das linhas na Figura 5.20 em que estão os pacotes nas matrizes Métodos e Eventos, respectivamente).

Como as matrizes utilizadas estão relacionadas a uma única execução, não está excluída a possibilidade de um pacote estar associado a uma característica apenas em função da ordem de execução. Esse é um dos pontos fracos de abordagens utilizando análise dinâmica. Para resolver essa situação seria necessário a comparação de várias execuções em ordens diferentes. Porém, considerando a execução do sistema por parte de um usuário, fica inviável cobrir todas as situações. Além disso, há casos em que não se pode alterar a ordem de execução em função da dependência entre as características ou passos de execução. Para remediar tal situação, uma das alternativas é comparar a participação dos pacotes entre as demais características. A existência de uma

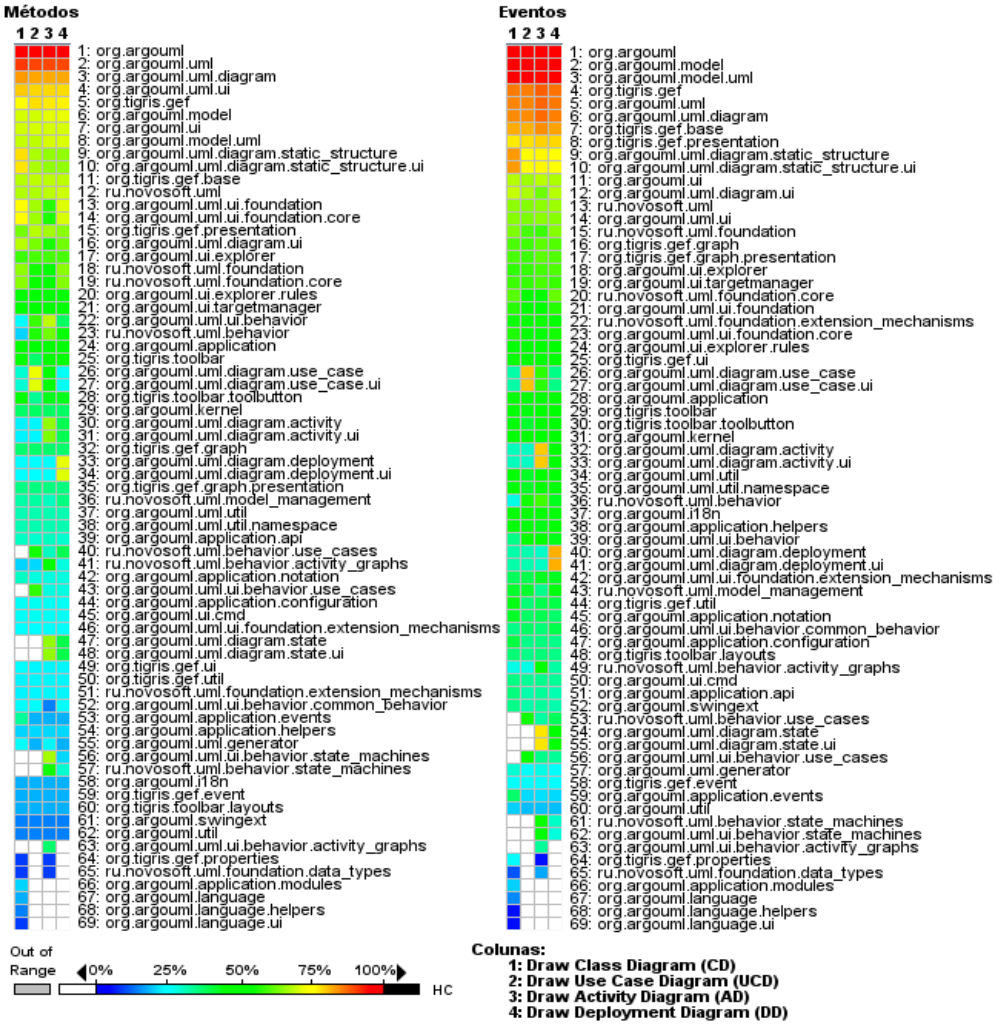


Figura 5.20: Participação em número de eventos e métodos nas características (Execução 1, Roteiro 1).

Tabela 5.25: Pacotes com maior concentração em cada característica.

Características	*Pacotes com maior concentração*
Draw Class Diagram (CD)	org.argouml.uml.diagram.static_structure (I.9,9), org.argouml.uml.diagram.static_structure.ui (I.10,10)
Draw Use Case Diagram (UCD)	org.argouml.uml.diagram.use_case (I.26,26), org.argouml.uml.diagram.use_case.ui (I.27,27)
Draw Activity Diagram (AD)	org.argouml.uml.diagram.activity (I.30,32), org.argouml.uml.diagram.activity.ui (I.31,33), org.argouml.uml.diagram.state (I.47,54), org.argouml.uml.diagram.state.ui (I.48,55), org.argouml.uml.ui.behavior.state_machines (I.56,61), org.argouml.uml.ui.behavior.activity_graphs (I.57,62)
Draw Deployment Diagram (DD)	org.argouml.uml.diagram.deployment (I.33,40), org.argouml.uml.diagram.deployment.ui (I.40,41)

discrepância pode indicar uma relação mais próxima com uma delas, como feito no parágrafo anterior para identificar os pacotes que se concentram mais em uma das características.

4.3.4.2 Análise de Classes

Ao contrário dos pacotes, várias classes são exclusivas de cada característica e são mostradas na Tabela 5.26. Como pode ser observado, o próprio padrão de

Tabela 5.26: Classes exclusivas das características.

Características	***Classes Exclusivas***
Draw Class Diagram	**org.argouml.language.helpers:** NotationHelper **org.argouml.uml.diagram.static_structure:** ClassDiagramGraphModel **org.argouml.uml.diagram.static_structure.ui:** ClassDiagramRenderer, SelectionClass, SelectionInterface, StylePanelFigClass, StylePanelFigInterface, PropPanelUMLClassDiagram **org.argouml.uml.ui.foundation.core:** PropPanelClass, PropPanelInterface, UMLClassActiveCheckBox, UMLClassAttributeListModel, UMLClassOperationListModel, UMLClassifierFeatureListModel, UMLModelElementVisibilityRadioButtonPanel, UMLNamespaceOwnedElementListModel, MElementResidenceImpl **ru.novosoft.uml.foundation.data_types:** MvisibilityKind
Draw Use Case Diagram	**org.argouml.model.uml:** UseCasesFactoryImpl **org.argouml.uml.diagram.ui:** PropPanelUMLUseCaseDiagram **org.argouml.uml.diagram.use_case:** UseCaseDiagramGraphModel **org.argouml.uml.diagram.use_case.ui:** SelectionActor, SelectionUseCase, StylePanelFigUseCase, UseCaseDiagramRenderer **org.argouml.uml.ui.behavior.use_cases:** PropPanelActor, PropPanelUseCase, UMLUseCaseExtendListModel, UMLUseCaseExtensionPointListModel, UMLUseCaseIncludeListModel
Draw Activitiy Diagram	**org.argouml.model.uml:** ActivityGraphsFactoryImpl, StateMachinesFactoryImpl **org.argouml.ui:** ActionAutoResize **org.argouml.uml.diagram.activity:** ActivityDiagramGraphModel: ActivityDiagramRenderer **org.argouml.uml.diagram.activity.ui:** SelectionActionState **org.argouml.uml.diagram.state.ui:** ActionCreatePseudostate **org.argouml.uml.diagram.ui:** PropPanelUMLActivityDiagram **org.argouml.uml.ui.behavior.activity_graphs:** PropPanelActionState UMLStateDoActivityList, UMLStateDoActivityListModel, UMLStateEntryList, UMLStateEntryListModel, UMLStateExitList, UMLStateExitListModel, UMLStateInternalTransition, UMLStateVertexContainerListModel, UMLStateVertexIncomingListModel, UMLStateVertexOutgoingListModel, UMLTransitionEffectList, UMLTransitionEffectListModel, UMLTransitionGuardListModel,UMLTransitionSourceListModel, UMLTransitionStateListModel, UMLTransitionStatemachineListModel, UMLTransitionTargetListModel, UMLTransitionTriggerList, UMLTransitionTriggerListModel **ru.novosoft.uml.behavior.state_machines:** MCompositeStateImpl **ru.novosoft.uml.foundation:** data_types.MPseudostateKind
Draw Deployment Diagram	**org.argouml.model:** RemoveAssociationEvent **org.argouml.uml.diagram.deployment:** DeploymentDiagramGraphModel **org.argouml.uml.diagram.deployment.ui:** DeploymentDiagramRenderer, SelectionComponent, SelectionNode **org.argouml.uml.diagram.ui:** PropPanelUMLDeploymentDiagram **org.argouml.uml.ui.foundation.core:** PropPanelComponent, PropPanelNode, UMLComponentResidentListModel, UMLContainerResidentListModel

nomes das classes mostra que elas tem uma relação muito próxima com as características e seus respectivos passos de execução. Vários conceitos específicos aos diagramas UML exercitados tem um correspondente nas classes apresentadas

na tabela, por exemplo, `SelectionClass` e `SelectionInterface` para *Classes* e *Interfaces* na característica *"Draw Class Diagram"*, `SelectionActor` e `SelectionUseCase` para *Ator* e *Caso de Uso* na característica *"Draw Use Case Diagram"*, `SelectionActionState` para Ação em *"Draw Activity Diagram"*, `SelectionComponent` e `SelectionNode` para *Componentes* e *Nó* em *"Draw Deployment Diagram"*.

4.3.4.3 Análise de Métodos

Há uma grande variedade de métodos exclusivos de cada característica. Por questões de espaço fica inviável a listagem de todos. Assim, optou-se por tabular apenas as classes contendo métodos exclusivos das características e que são mostradas nas Tabelas 5.27 e 5.28. Após o nome da característica está o número de métodos exclusivos associados, por exemplo, 533 para "*Draw Class Diagram".*

Tabela 5.27: Classes com métodos exclusivos.

Características	*Classes com métodos exclusivos*
Draw Class Diagram (533)	**org.argouml.uml.ui.foundation.core:** PropPanelInterface, UMLClassActiveCheckBox, UMLClassAttributeListModel, UMLClassifierFeatureListModel, UMLModelElementVisibilityRadioButtonPanel, UMLNamespaceOwnedElementListModel, **org.tigris.gef.base:** SelectionManager **org.tigris.gef.presentation:** FigRect, FigText **org.tigris.toolbar.toolbutton:** PopupToolBoxButton **ru.novosoft.uml:** MFactoryImpl **ru.novosoft.uml.foundation.core:** MClassImpl, MElementResidenceImpl, MInterfaceImpl, **ru.novosoft.uml.foundation.data_types:** MvisibilityKind
Draw Use Case Diagram (411)	**org.argouml.model.uml:** DefaultModelImplementation, NSUMLModelFacade, UmlFactoryImpl **org.argouml.uml.diagram.ui:** ActionAddExtensionPoint, PropPanelUMLUseCaseDiagram **org.argouml.uml.diagram.use_case:** UseCaseDiagramGraphModel **org.argouml.uml.diagram.use_case.ui:** FigActorFigUseCase$FigMyCircle, FigUseCase, SelectionActor, SelectionUseCase, StylePanelFigUseCase, UMLUseCaseDiagram **org.argouml.uml.ui.behavior.use_cases:** ActionNewActor, ActionNewUseCaseExtensionPoint, PropPanelActor, PropPanelUseCase$ActionNewExtensionPoint, PropPanelUseCase$ActionNewUseCase, PropPanelUseCase, UMLUseCaseExtendListModel, UMLUseCaseExtensionPointListModel, UMLUseCaseIncludeListModel, FigCircle, FigGroup, FigLine, FigRect, MFactoryImpl, MActorImpl, MuseCaseImpl

Tabela 5.28: Classes com métodos exclusivos.

Características	*Classes com métodos exclusivos*
Draw Activity Diagram (633)	**org.argouml.application.api:** Notation **org.argouml.model.Model:** getPseudostateKind **org.argouml.model.uml:** ActivityGraphsFactoryImpl, CoreHelperImpl, DefaultModelImplementation, KindsImpl, MetaTypesImpl, NSUMLModelFacade, StateMachinesFactoryImpl, StateMachinesHelperImpl, UmlFactoryImpl **org.argouml.ui:** ActionAutoResize **org.argouml.uml.diagram.activity:** ActivityDiagramGraphModel **org.argouml.uml.diagram.activity.ui:** ActivityDiagramRenderer, FigActionState, SelectionActionState, UMLActivityDiagram, ActionCreatePseudostate, ActionCreatePseudostate, FigInitialState, FigTransition **org.argouml.uml.diagram.ui:** PropPanelUMLActivityDiagram **org.argouml.uml.generator:** GeneratorDisplay, **org.argouml.uml.ui.behavior.activity_graphs:** PropPanelActionState **org.argouml.uml.ui.behavior.state_machines:** ActionNewGuard, ActionNewTransition **org.argouml.uml.ui.behavior.state_machines:** PropPanelPseudostate, PropPanelTransition, UMLStateDeferrableEventListModel, UMLStateDoActivityList, UMLStateEntryList, UMLStateEntryListMode, UMLStateExitList, UMLStateExitListModel,UMLStateInternalTransition, UMLStateVertexContainerListModel, UMLStateVertexIncomingListModel, UMLStateVertexOutgoingListModel, UMLTransitionEffectList, UMLTransitionGuardListModel, UMLTransitionSourceListModel, UMLTransitionSourceListModel, UMLTransitionStateListModel, UMLTransitionStatemachineListModel, UMLTransitionTargetListModel, UMLTransitionTriggerList, UMLTransitionTriggerListModel **org.tigris.gef.presentation:** FigCircle, FigRRect **ru.novosoft.uml:** MFactoryImpl **ru.novosoft.uml.behavior.activity_graphs:** MActionStateImpl, MActivityGraphImpl **ru.novosoft.uml.behavior.state_machines:** MCompositeStateImpl, MCompositeStateImpl,MPseudostateImpl, MTransitionImpl, MTransitionImpl **ru.novosoft.uml.foundation.data_types:** MpseudostateKind
Draw Deployment Diagram (375)	**org.argouml.model:** RemoveAssociationEvent **org.argouml.model.uml:** CoreFactoryImpl, ExplorerNSUMLEventAdaptor, MetaTypesImpl, NSUMLClassEventListener, NSUMLModelEventListener, NSUMLModelFacade, UmlModelEventPump **org.argouml.ui.explorer:** ExplorerTreeModel **org.argouml.uml.diagram.deployment:** DeploymentDiagramGraphModel **org.argouml.uml.diagram.deployment.ui:** DeploymentDiagramRenderer, FigComponent, FigMNode, SelectionComponent, SelectionNode, UMLDeploymentDiagram org.argouml.uml.diagram.ui: PropPanelUMLDeploymentDiagram org.argouml.uml.ui.foundation.core; PropPanelComponent, PropPanelNode, UMLComponentResidentListModel **org.tigris.gef.presentation:** FigCube, FigRect ru.novosoft.uml: MFactoryImpl **ru.novosoft.uml.foundation.core:** MComponentImpl, MDependencyImpl, MNodeImpl **ru.novosoft.uml.model_management:** MmodelImpl

4.3.5 Análise de Similaridade de Implementação entre Características

A similaridade entre as características pode ser observada através das matrizes Característica x Característica. A Figura 5.21 ilustra três expansões sucessivas usando esse tipo de matriz. A escala usada é a *Hot Cold* e a medida de similaridade entre as características é o número de métodos comuns entre elas, com base na execução do Roteiro 1. A primeira matriz mostra apenas as características principais relacionadas ao desenho dos diagramas. De acordo com a matriz, essas características tem um grau de compartilhamento de métodos que fica em torno de 50%, o que pode ser atestado pela coloração média das células (verde) posicionada

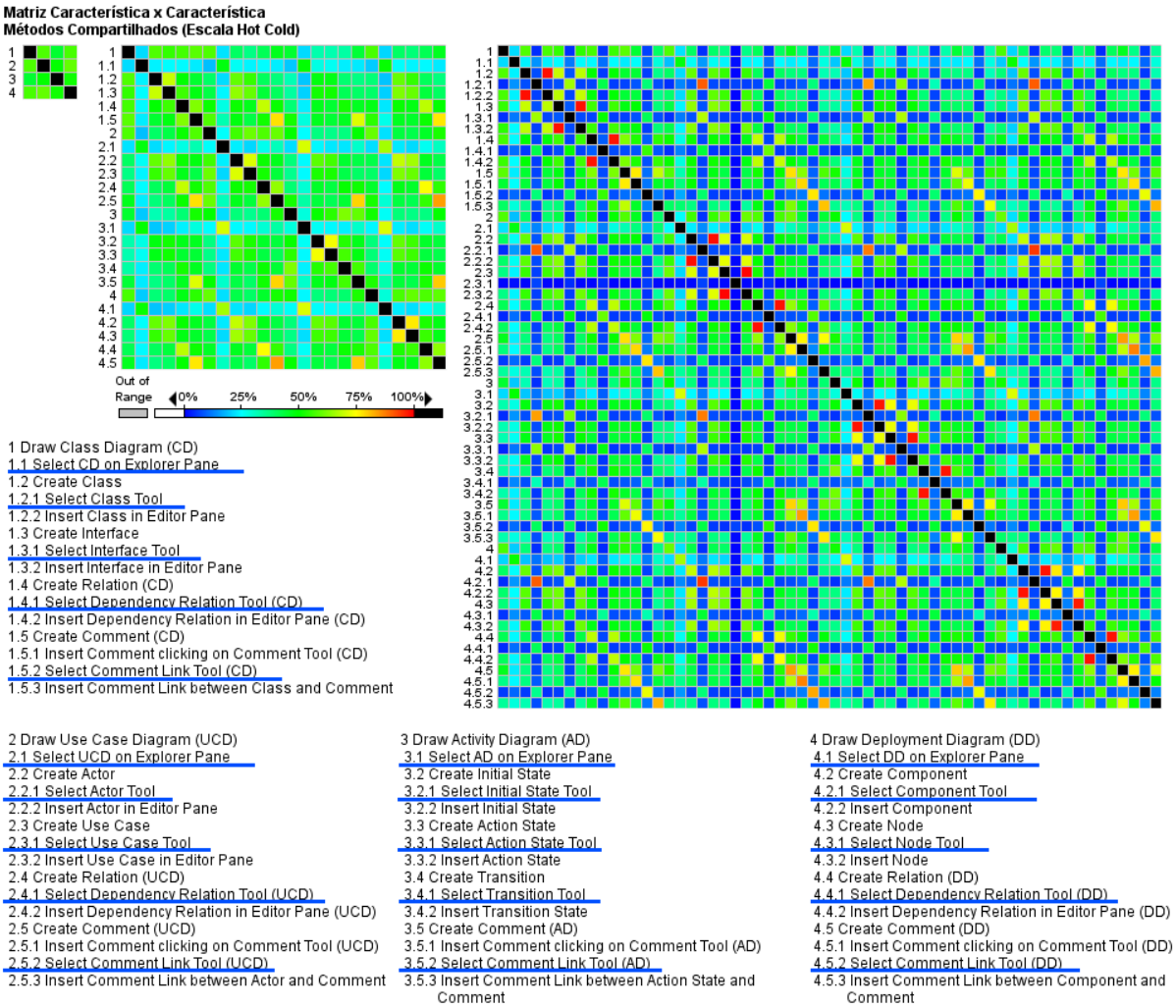


Figura 5.21: Similaridade entre características observada em expansões sucessivas das matrizes Característica x Característica.

no meio da escala *Hot Cold*. A segunda matriz representa a primeira expansão realizada. Dessa vez algumas células com coloração diferenciada aparecem pela matriz, mostrando as primeiras variações de similaridade entre as características. Pode ser observado também a formação de algumas “cruzes” pela matriz, indicadas por linhas e colunas com coloração azul quase uniforme e que contrasta com a coloração verde na maior parte das células da matriz. Essas cruzes remetem ao padrão *Dotplot “Light Cross”* (Capítulo 2 – Referencial Teórico, Seção 6.3.2). De acordo com o padrão os elementos ao longo da diagonal que estão no centro da cruz, representam elementos pouco relacionados com os demais. É interessante notar que nos pontos de cruzamento das cruzes a célula correspondente tem uma tonalidade mais forte (verde ou amarelo) que as demais células formando a cruz. Isso indica que embora os elementos que originam as cruzes tenham uma relação

mais fraca com os demais elementos na matrizes, eles estão relacionados entre si (em outras palavras compartilham um número maior de métodos). A expansão seguinte da matriz resulta em mais cruzes espalhadas pela matriz, com uma tonalidade ainda mais fraca (azul escuro), indicando que esses elementos são ainda menos relacionados com os demais. Curiosamente, todos os elementos que forma as cruzes azuis ao longa da matriz estão relacionados aos passos de execução de seleção das funcionalidades (ver elementos sublinhados na legenda das linhas e colunas da matriz na Figura 5.21). Esse fato mostra que a interpretação do padrão *"Light Cross"* é consistente, pois todas as cruzes são originadas de elementos conceitualmente similares e que, de acordo com a matriz compartilham grande parte de elementos do código.

Sabendo que os passos de execução relacionados à seleção de funcionalidades tem vários métodos em comum entre si, mas poucos com os demais passos de execução ou características, optou-se por removê-los da matriz para continuar a análise. Após a remoção desses elementos, a matriz resultante deixou mais clara a presença de um segundo padrão *Dotplot*, o padrão *Diagonals*, que fica mais evidente na escala de cores *Discrete Levels*. As matrizes são mostradas na Figura 5.22, usando a escala *Hot Cold* e *Discrete Levels*, respectivamente. Os limites correspondentes à criação de cada diagrama são mostrados ao longo da diagonal principal. Pode ser visto que ao longo das diagonais secundárias (fora da diagonal principal), existe uma tonalidade um pouco mais forte nas células, indicando uma compartilhamento maior de métodos entre os elementos correspondentes. O destaque é mais evidente especialmente nas células das linhas 1.4. 1.4.1, 1.4.2, 2.4, 2.4.1, 2.42, 3.4, 3.4.1, 3.4.2, 4.4, 4.4.1, 4.4.2. Essas linhas correspondem a criação dos comentários nos diagramas e são os passos de execução recorrentes em cada diagrama. Há ainda algumas células um pouco mais evidentes nas linhas 1.3, 1.31, 2.3, 2.3.1, 4.3, 4.3.1. Essas são as linhas referentes à inserção de relações de dependências que ocorrem em três casos apenas (Diagrama de Classes, Diagrama de Casos de Uso e Diagrama de Implantação). Nota-se que para o *Diagrama de Atividades* nas linhas (e colunas) correspondentes 3.3 e 3.3.1 não se destacam como as citadas, por serem referentes à inserção de uma transição (*Create Transition*) e não de uma relação de dependência. Esses dois casos mostram que há reúso de métodos na criação dos diagramas, especialmente na inserção de comentários, o que era esperado num sistema bem implementado.

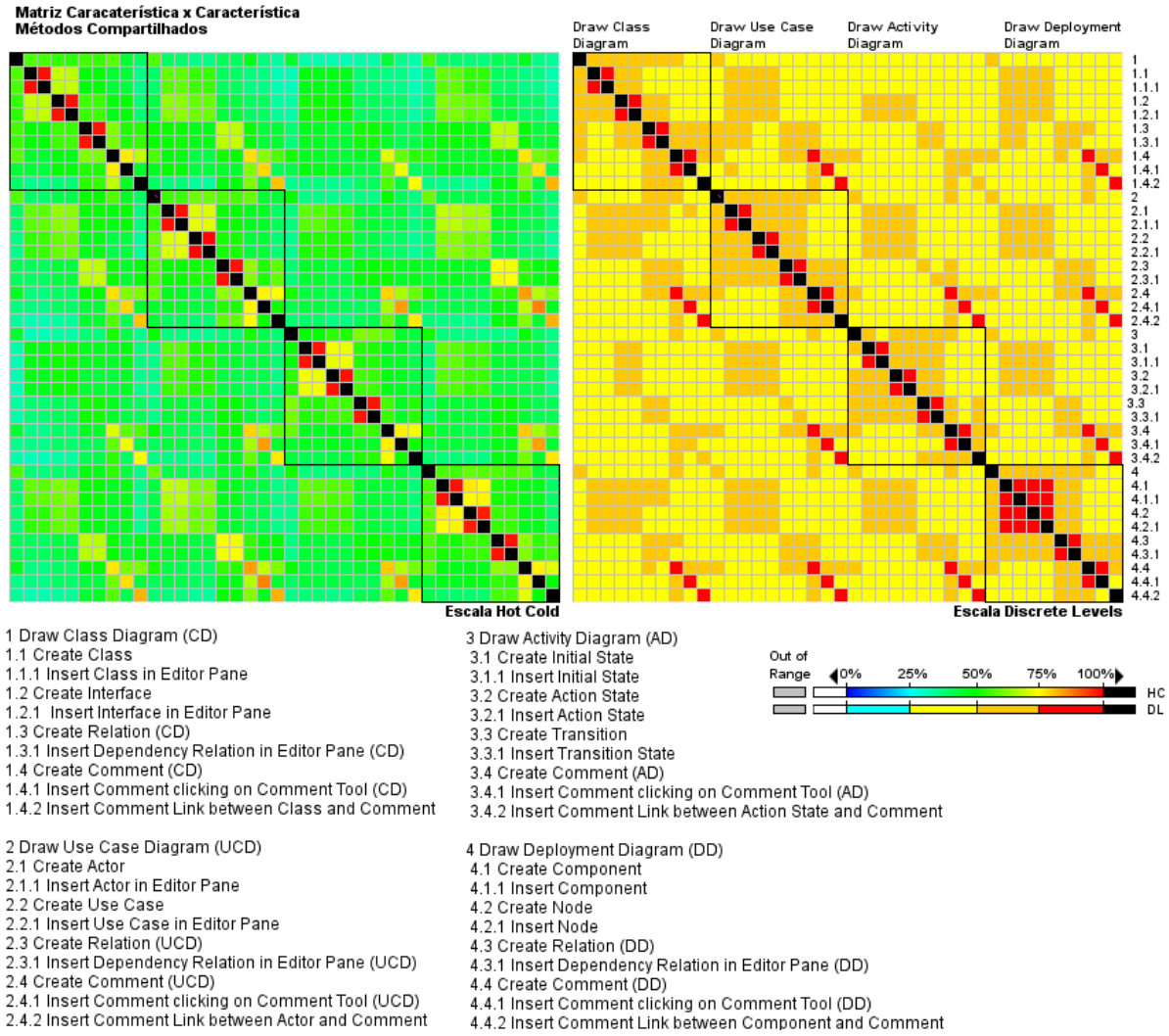


Figura 5.22: Similaridade entre características destacando na diagonal principal as células relacionadas à criação de cada diagrama.

Através da seleção da célula e detalhamento das informações nas matrizes mostradas na Tabela 5.22, é possível ver os métodos compartilhados pelas características e conseqüentemente suas respectivas classes. Dentre as 208 classes comuns na inserção de comentários e as 211 comuns na inserção de dependências, obtidas após o detalhamento das métricas pela seleção das células, as classes mostradas na Tabela 5.29 merecem destaque especial, tanto pela correlação de seus nomes com os nomes das características (inserção de comentário e inserção de dependências) quanto pelo número mais expressivo de métodos utilizados nos respectivos passos de execução.

Tabela 5.29: Classes relacionadas à execução de dois passos de execução comuns entre o desenho de diagramas de classe.

Passo de Execução	*Classes relacionadas*
Insert Comment Clicking on Comment Tool (CD, UD, AD, DD)	org.argouml.uml.diagram.static_structure.ui: FigComment, SelectionComment, org.argouml.uml.diagram.ui:ActionAddNote org.argouml.uml.ui.foundation.core: PropPanelComment, UMLCommentAnnotatedElementListModel, UMLCommentBodyDocument, org.tigris.gef.presentation: FigPoly, FigRect, FigText ru.novosoft.uml.foundation.core: McommentImpl
Insert Dependency Relation on Editor Pane (CD, UD, DD)	org.argouml.uml.diagram.ui: FigDependency org.argouml.uml.ui.foundation.core: PropPanelDependency, UMLDependencyClientListModel, UMLDependencySupplierListModel org.tigris.gef.presentation: FigPoly, FigText ru.novosoft.uml.foundation.core.MDependencyImpl

4.3.6 Redução do escopo para análise e compreensão do sistema

Como realizado no Contexto 1, a redução no escopo será analisada comparando métricas do código fonte com métricas extraídas dos rastros de execução. A Figura 5.23 mostra os resultados obtidos. Mais uma vez, a redução no

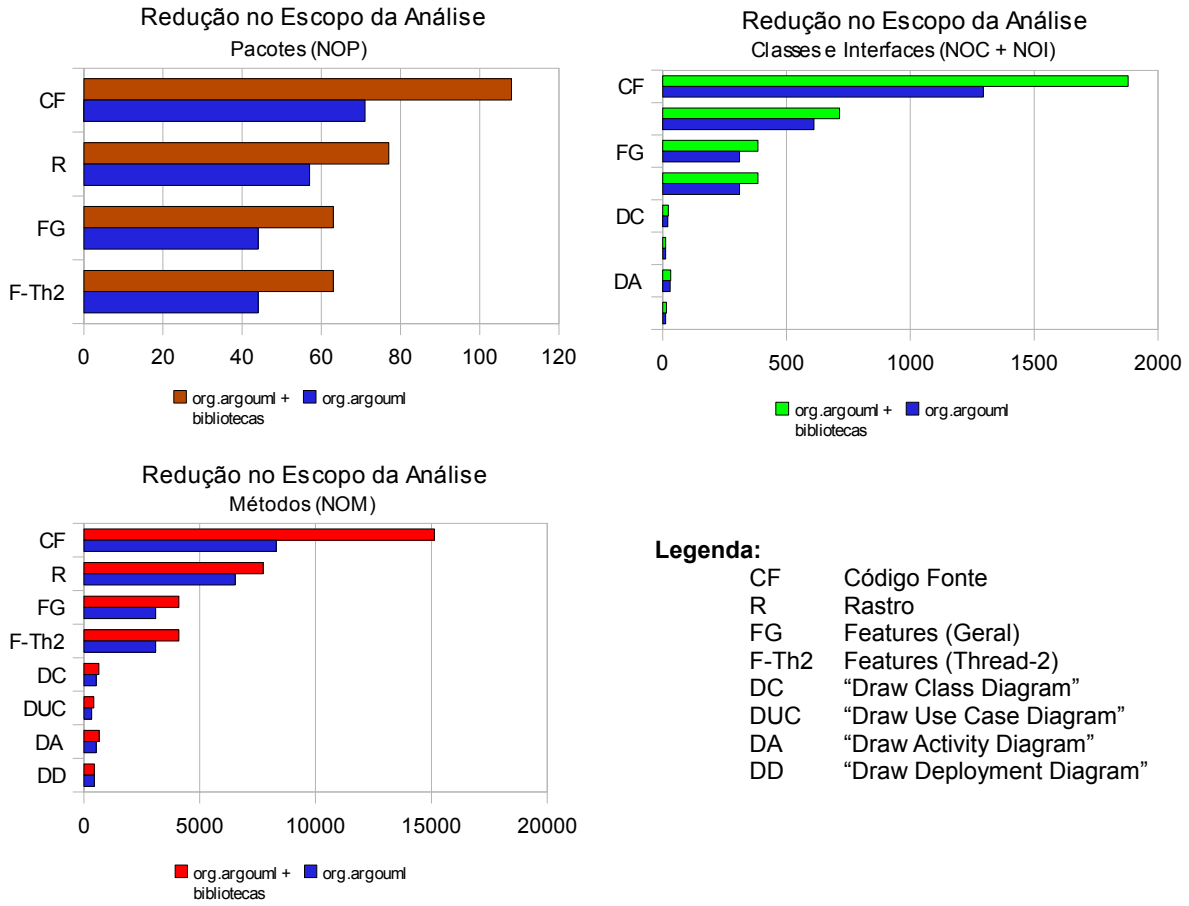


Figura 5.23: Redução no escopo para a análise de Pacotes, Classes e Métodos.

número de elementos a se considerar é grande tanto em relação ao número de elementos no código quanto ao número de elementos no rastro. Como as características não se manifestaram em outras *threads* além da *Thread-2*, o número de elementos associados às características é o mesmo para todo o rastro e para a *Thread-2*. Os valores mostrados por cada características referem-se ao número de elementos exclusivos dessas características.

4.3.7 Influência da ordem de execução sobre as análises

Como as características no Contexto 2 possuem diferenças mais significativas (ao menos conceitualmente), a influência da ordem de execução foi evidenciada nos resultados na análise de forma mais expressiva que no Contexto 1. A impressão inicial é de que quando uma característica é executada, as características sub-seqüentes acabam sendo associadas a alguns elementos da característica anterior, mesmo que não haja alguma relação efetiva entre elas. Esse tipo de situação foi observada especialmente para elementos da interface gráfica ou que sejam acessados continuamente por ela, para manter alguma visualização atualizada.

Para ilustrar, os pacotes presentes no rastro e associados aos *frameworks* utilizados pelo *ArgoUML* são analisados. A Figura 5.24 mostra as matrizes com a métrica NSEPF e na escala *Hot Cold Log* geradas com base no rastro da primeira e segunda execução dos roteiros 1 e 2. Pode ser observado que entre as execuções do mesmo roteiro, as matrizes são praticamente idênticas, o que demonstra a consistência entre as execuções. No entanto, as linhas 9, 18 e 19 são diferentes entre os roteiros de execução. A linha 9 indica que o pacote `org.tigris.gef.properties` participa nas colunas 1 e 3, para o Roteiro 1, e nas colunas 3 e 4, para o Roteiro 2. Nesse caso a inconsistência é evidenciada entre as colunas 1 e 4. É importante notar que no Roteiro 1, a característica na coluna 1 é a primeira a ser executada, enquanto a característica na coluna 4 é a última. No Roteiro 2, essa ordem é invertida. Com base nisso, pode-se deduzir que o pacote `org.tigris.gef.properties` tenha relação com a característica que executa primeiro e não com a característica em si. Um segundo caso acontece nas linhas 18 e 19. Novamente há uma diferença entre as colunas 1 e 4, para as matrizes de cada roteiro. Porém, dessa vez há uma inversão na ordem. Os pacotes participam na coluna 4 para o primeiro roteiro e na coluna 1 para o segundo. Como a característica

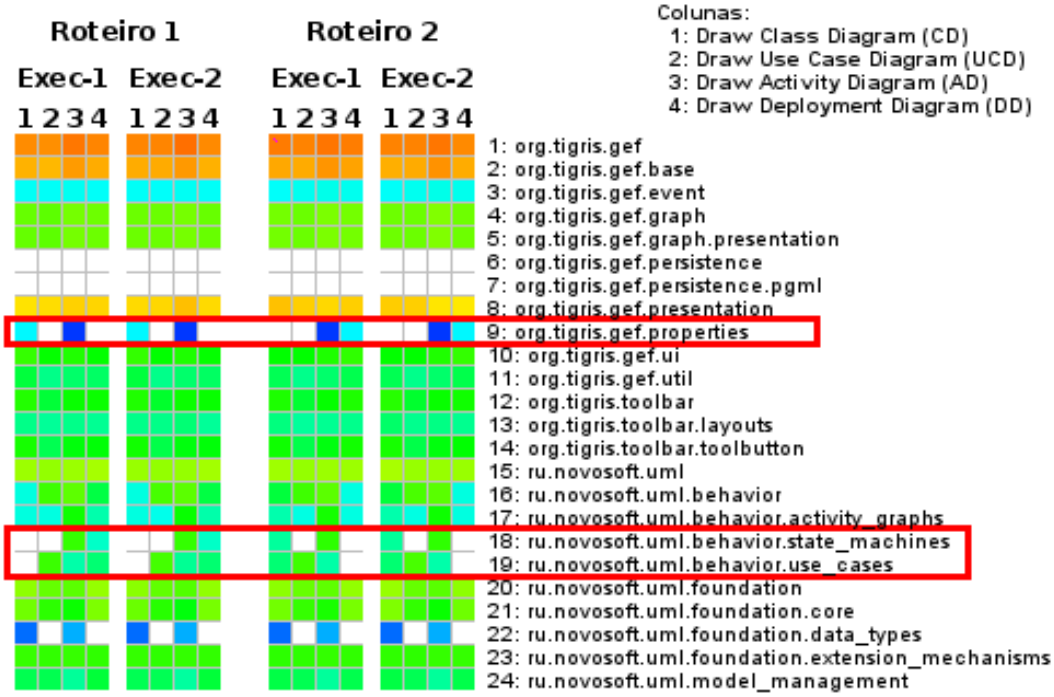


Figura 5.24: Influência da ordem de execução sobre a interpretação das matrizes (métrica NSEPF e escala *Hot Cold Log*).

da coluna 1 foi a primeira a ser executada no Roteiro 1 e não mostrou relação com os pacotes das linhas 18 e 19, ela não deve depender desses pacotes, pois só foi acusada alguma participação quando a característica foi a última a ser executada. O mesmo raciocínio vale para a característica na coluna 4, porém considerando a inversão na ordem de execução.

A conclusão é que a relação entre os pacotes citados e as características só se estabeleceu em função da ordem de execução. Situações similares às descritas foram verificadas também no nível de classes e métodos. Isso mostra duas coisas: 1. a ordem de execução definida para o exercício das características nos roteiros realmente tem influencia nas análises; 2. os objetos criados a partir dessas classes, continuam sendo acessados após sua criação, mesmo que não haja uma relação direta deles com a característica sendo exercitada. Assim, com base na implementação corrente não é possível afirmar a partir de um único rastro de execução que uma característica depende incondicionalmente de uma classe ou método, quando mais características estão envolvidas e essas características foram exercitadas anteriormente. Só é possível confirmar essa dependência quando o elemento está associado exclusivamente a essa característica ou quando a característica foi a primeira a ser executada. Além disso, é preciso estar atento às

funcionalidades que ficam em execução contínua no sistema (como ocorre com as "Auto Críticas").

4.4 Discussão

Este estudo mostrou novamente o potencial da abordagem em relacionar características e elementos do código com base em roteiros de execução. Como no Contexto 1, foi mostrado que a quantidade de elementos do código efetivamente associados às características é muito menor que o universo de elementos, tanto em nível de código fonte, quanto com relação ao total de elementos ativados no rastro de execução.

Ao contrário do Contexto 1, as diferenças entre os elementos participantes nas características ficaram mais evidentes. Isto remete ao cuidado que deve ser tomado na escolha dos roteiros de execução. No Contexto 1, por serem muito similares, tornou-se difícil distinguir elementos específicos das características. No Contexto 2, vários desses elementos ficarão evidentes, especialmente para classes e métodos que mostraram vários itens específicos nas características.

A ordem de execução das características revelou-se como um problema na abordagem. Porém, este é um fato compartilhado entre praticamente todas as a abordagens similares. A detecção do problema necessita a execução de roteiros com ordens diferentes de execução nos passos. Os rastros gerados devem então ser comparados. Em função da grande quantidade de elementos envolvidos, é desejável que haja suporte ferramental para isso.

5 Problemas e limitações da abordagem

Quando uma característica é executada e influencia na execução das posteriores apenas por ter sido executada antes, está caracterizado um dos problemas da abordagem: a influência da ordem de execução. Estes eventos de "fundo" podem causar enganos na interpretação dos resultados, levando o analista a associar a execução de uma característica a elementos do código sem relação real. A mudança na ordem de execução e comparação de diferentes rastros pode ser uma alternativa, porém, nem sempre viável. Em alguns casos, devido a

dependências entre as características a ordem de execução não pode ser alterada. Por exemplo, a remoção de um elemento do modelo de um diagrama deve ser precedida pela criação desse elemento, portanto, não há como mudar essa ordem, embora possa ser interessante analisar as características em separado. Mesmo que seja possível alterar a ordem, pode haver uma explosão combinatória, dependendo do número de passos de execução ou características utilizado. Assim, cobrir todas as possibilidades torna-se inviável dado que a abordagem supõe a presença de um usuário para a captura das informações. Uma alternativa para amenizar o problema, poderia ser a utilização de casos de teste, quando possível, que automatizassem a entrada do usuário.

O tratamento de rastros múltiplos é outra questão a ser considerada. A adição dessa dessa opção permitiria a análise e comparação de diferentes execuções. Isto poderia ser feito para o mesmo sistema ou para vários (incluindo diferentes versões do mesmo sistema). Essa opção poderia facilitar também a identificação de inconsistências geradas por influência da ordem de execução.

A compressão do rastro é uma das opções para lidar com o grande volume de dados envolvido nos rastros. Porém, estudos mais detalhados precisam ser realizados sobre os efeitos na análise de se utilizar os rastros originais ou os rastros comprimidos e a ferramenta também precisa ser adaptada para fazer a distinção entre eles. Essa foi a razão principal que levaram ao uso dos rastros originais no estudos realizados nesse trabalho. O cálculo sobre demanda e o armazenamento em cache também são fundamentais para lidar com o volume de dados. A ferramenta pode ter uma melhora considerável, em especial, no tempo de resposta para o usuário se algoritmos e estruturas de dados mais eficientes forem utilizados.

CAPÍTULO 6. TRABALHOS RELACIONADOS

1 OBJETIVOS

Neste capítulo alguns trabalhos relacionados serão discutidos. Os trabalhos mostram outras abordagens para a localização de características. Além da Análise Dinâmica, várias outras técnicas são combinadas nestes trabalhos tais como Análise Estática, Técnicas de Recuperação de Informação, Ranking Probabilístico, Ranking Heurístico, Análise de Impacto, Análise de Métricas e Análise Formal de Conceitos. Para facilitar a descrição das comparações, os comentários referentes a este trabalho serão indicados pelo nome da ferramenta desenvolvida, *Featincode*.

2 OUTRAS ABORDAGENS PARA A LOCALIZAÇÃO DE CARACTERÍSTICAS

2.1 SALAH E OUTROS

Uma abordagem para a compreensão de sistemas de grande porte é proposta por Salah e Mancoridis (Salah {et al}, 1996) com base em análise dinâmica e dirigida por roteiros de uso. Para isso, visões em diferentes perspectivas são geradas. Os autores também apóiam-se na idéia de que as requisições de mudança sobre um sistema são escritas em linguagem natural com referências explícitas às características dos sistema. Dessa forma, busca-se viabilizar ao desenvolvedor o mapeamento automático entre requisições de mudanças e porções do código fonte relacionadas às características, ao invés de tentar fazer esse mapeamento de forma manual, através na análise direta do código. Os roteiros de uso podem ser obtidos das descrições de casos de uso e requisições de mudança. A análise dinâmica é realizada sobre rastros de execução gerados com base nesses roteiros e demarcados pelo analista, de maneira similar a *Featincode*.

Dois tipos de perspectivas sobre o sistema são produzidas: 1. visões em

forma de grafos; 2. métricas de similaridade entre os casos de uso. As visões em grafo mostram as relações em tempo de execução entre entidades em diferentes níveis de abstração (métodos, classes, módulos e casos de uso). Por sua vez, as métricas de similaridade apontam o grau de proximidade dos casos de uso em termos de elementos compartilhados na sua implementação. Como em *Featincode*, a medida de similaridade entre casos de uso é apresentada por meio de matrizes de similaridade. Porém, a medida de similaridade é fornecida numericamente usando a métrica *Jaccard Index*. Em *Featincode,* a coloração das células das matrizes é usada para indicar o grau de similaridade entre os elementos, os quais não se restringem apenas a casos de uso (equivalente às características) mas também a pacotes, classes e métodos. Além disso, *Featincode* pode ser estendido para suportar a apresentação numérica das métricas ou mesmo para o uso de novos esquemas de cores.

O trabalho lida ainda com a filtragem de elementos para reduzir a confusão visual inerente a grande quantidade de elementos e relações de sistemas de grande porte, bem como o agrupamento desses elementos. A filtragem pode ser feita pelo próprio esquema de navegação ou com base nas relações entre as entidades e os casos de uso (Ex.: número de características que um elemento está associado). Os mecanismos de agrupamento são baseados nas medidas de coesão e acoplamento das chamadas entre as entidades e destacados no disposição dos grafos. Em *Featincode* o agrupamento é feito com base nas métricas calculadas e visualizados pela coloração das células das matrizes. Além disso, o mecanismo de filtragem permite remover itens nas matrizes com base em padrões de nomes, tipo do elemento (pacote, classe, método, característica) e valor das métricas.

Enquanto o trabalho foca na visualização de grafos mostrando relacionamentos entre as entidades com base nas chamadas entre elas, *Featincode* gera visualizações matriciais com três focos: 1. o relacionamento entre elementos do código e características; 2. a similaridade entre as elementos do código com base na quantidade de características compartilhadas; 3. a similaridade entre as características com base na quantidade de elementos do código compartilhados.

2.2 ANTONIOL E GUÉHÉNEUC

O trabalho de Antoniol e Guéhéneuc (ANTONIOL; GUÉHÉNEUC, 2005)

apresenta uma forma de identificação e localização de características para programas de grande porte, com processamento concorrente, orientados a objetos (dando suporte a programas C++). Para superar as dificuldades encontradas em abordagens baseadas apenas em análise dinâmica, o trabalho combina diferentes técnicas tais como análise estática e dinâmica, filtragem de conhecimento, e ranking probabilístico. Um modelo de transformações é usado para comparar e visualizar as características identificadas. Os dados são coletados pela execução de uma funcionalidade do sistema com base em diferentes roteiros de execução. A análise dinâmica serve como filtro para a análise estática. Modelos estáticos, como diagramas de classes, são então gerados e destacam diferenças entre as características.

As características servem como ponto de ligação entre um modelo de arquitetura do programa e o seu comportamento dinâmico. O modelo de arquitetura é construído primeiramente com o auxílio da análise estática do código fonte. A seguir as características são identificadas para prover aos mantenedores subconjuntos da arquitetura (ou micro-arquiteturas) representando classes, métodos e campos que participam nas características de interesse. Finalmente, características diferentes (as micro-arquiteturas) para destacar as diferenças.

As características são identificadas por análise dinâmica a partir da execução de roteiros. Os rastros são modelados como seqüência de intervalos, que são seqüências de eventos (criação de um objeto, execução de um fragmento do código fonte). Dois tipos de roteiros são considerados: aqueles relacionados à característica de interesse e aqueles que não exercitam a característica. Cada roteiro é executado para a coleta dos rastros. Intervalos são marcados nestes roteiros como relevantes ou não para a característica, usando ranking probabilístico (um ranking quantificando a probabilidade de que um evento seja relevante para a característica). Eventos relevantes sempre serão apresentados nos intervalos relevantes da característica.

A filtragem da base de conhecimento é necessária para remoção dos eventos irrelevantes para os roteiros de interesse. Isto ajuda na redução da quantidade de dados dinâmicos, auxiliando no processo de entendimento do programa.

O modelo de característica é construído com base nas micro-arquiteturas, incluindo somente classes, métodos e campos ativados explicitamente ao executar o roteiro que exercita a característica de interesse.

A comparação entre as características é feita destacando-se as diferenças

entre as micro-arquiteturas com o objetivo de permitir aos mantenedores compreender e comparar o comportamento de funcionalidades diferentes ou da mesma funcionalidade com roteiros distintos. Para destacar estas diferenças é calculado o conjunto de transformações no modelo que são necessárias para transformar uma micro-arquitetura em outra.

Esta proposta permite a ordenação, com índice de relevância, de classes e métodos que contribuem para uma micro-arquitetura implementando uma característica. Além disso, permite identificar diferenças entre características precisamente.

Assim como este trabalho, *Featincode* tem o objetivo de auxiliar os desenvolvedores no processo de manutenção de software. *Featincode* utiliza apenas análise dinâmica para a localização e a comparação de características a partir de roteiros de execução, ao contrário de Antoniol e Guéhéneuc que utiliza também análise estática e ranking probabilístico. A comparação entre características em *Featincode* é feita com a utilização de matrizes de coloração, uma forma de visualização amigável ao usuário, que mostra a participação dos elementos do código fonte nas características, obtidas a partir do rastro de execução, e sendo feita uma comparação pontual entre os elementos. Já no trabalho de Antoniol, a comparação é feita através de probabilidade, que define o quão relevante um elemento é ou não para as características.

2.3 ZHAO E ZHANG

Zhao e Zhang propõe a abordagem *SNIAFL* (*Towards a Static Non-Interactive Approach to Feature Location*) (ZHAO et al 2004), a tarefa de localização de características é feita com base em uma abordagem estática e não interativa. A tecnologia de recuperação de informação (RI) é usada para revelar as conexões básicas entre características e unidades computacionais no código fonte. Uma representação estática do código fonte (*BRCG*) é utilizada para recuperar unidades relevantes e específicas computacionalmente para cada característica. A premissa de que os programadores usam nomes significativos como identificadores deve ser considerada para o sucesso da abordagem.

A abordagem *SNIAFL* foi validada com um sistema escrito em linguagem C. O foco de concentração foi a localização de funções (unidades computacionais)

relevantes e funções específicas de uma característica. Funções específicas são aquelas utilizadas na implementação de uma característica exclusivamente. Funções relevantes são as que estão envolvidas na implementação da característica, mas não são necessariamente exclusivas. O objetivo de SNIAFL é resolver o problema da localização de característica estaticamente, para isso a idéia básica é usar RI para revelar as conexões entre características e funções.

As características são descritas em linguagem natural e nomes significativos para os identificadores devem ter sido usados no código fonte. Um conjunto inicial de funções específicas de cada característica é buscado através de técnicas de RI que se baseiam no casamento entre as descrições dessas funções com os identificadores no código. Este conjunto de funções específicas é utilizado como base para a descoberta do conjunto de funções relevantes. Por fim, o conjunto final de funções relevantes é então encontrado. O processo pode ser resumido em quatro passos: 1 – Obter as conexões iniciais específicas entre característica e funções; 2 – Obter as funções iniciais específicas; 3 – Determinar funções e rastros relevantes e 4 – Determinar as funções específicas finais. Para o passo 1 utiliza-se RI para filtrar as informações especificas das características e recuperar as conexões iniciais. O segundo passo é ordenar as funções em cada característica de acordo com o resultado da recuperação e escolher a função específica inicial para cada característica. O terceiro passo é obter as funções relevantes e o possível pseudo rastro de execução usando BRCG, a partir do código fonte. O último passo é a análise das funções relevantes para determinar as funções específicas finais.

Comparando-se SNIAFL com *Featincode* tem-se que ambos surgem com o objetivo de facilitar as atividades de manutenção pela localização de características no código fonte. SNIAFL trabalha com sistemas desenvolvidos na linguagem C e *Featincode* na linguagem Java. O primeiro usa apenas abordagem estática combinada a técnicas de RI. A abordagem *Featincode* utiliza análise dinâmica e análise visual baseada em matrizes de visualização. Quanto à granularidade da análise, enquanto SNIAFL trabalha no nível de funções, *Featincode* trabalha com pacotes, classes e métodos.

2.4 ROHATGI E HAMOU-LHADJ

O trabalho de Rohatgi e Hamou-Lhadj (ROHATGI; HAMOU-LHADJ; RILLING,

2008) apresenta uma forma de localização de características que combina as técnicas de análise estática e dinâmica. Um rastro de execução é gerado através do exercício da característica estudada (análise dinâmica). Um gráfico de dependência (análise estática) é usado para ordenar os componentes invocados no rastro medindo o impacto da modificação do componente no resto do sistema (de acordo com a sua relevância na característica). Esta proposta baseia-se na hipótese de que quanto menor o impacto da modificação de um componente, mais provável é que este componente seja específico para uma característica. A vantagem deste trabalho está no fato de que requer somente a geração de um rastro de execução, o único correspondente à característica estudada, não requerendo que os usuários tenham um conhecimento extenso a priori do sistema.

A localização das características neste trabalho é feita em três passos: 1 – Geração do rastro de execução, de onde classes distintas são extraídas (perfil de execução) ; 2 – Utilização de métricas de análise de impacto (identificação das parte do programa que são potencialmente afetadas por uma mudança no programa) nas classes extraídas (baseados no gráfico de dependência estática das classes); 3 - Ordenação das classes para identificar classes especificas da característica.

As métricas desenvolvidas e utilizadas no passo 2 são: *TWI* (*Two Way Impact*) e *WTWI* (*Weighted Two Way Impact*). A métrica *TWI* considera o impacto aferente de uma classe C (número de classes que são afetadas quando a classe C é modificada) e o impacto eferente (número de classes que afetarão C se forem modificadas). A métrica *WTWI* usa informação disponível sobre a arquitetura do sistema, melhorando os resultados de *TWI*, e considera o pacote de impacto aferente da classe (número de pacotes afetados por uma modificação na classe C) . Para o cálculo dessas métricas é utilizado o gráfico de dependência de classes (*CDG*) obtido com a análise estática do código fonte.

Os resultados obtidos através destas métricas são utilizados na ordenação das classes (ordenados baseados nas classes especificas da característica – *TWI* e *WTWI* em ordem ascendente).

O trabalho de Rohatgi e Hamou-Lhadj difere-se da abordagem do *Featincode* por combinar análise estática e dinâmica, enquanto *Featincode* utiliza apenas a análise dinâmica. Os dois trabalhos compartilham o mesmo objetivo inicial, a criação de um método para auxiliar os mantenedores de software na localização de características. As duas abordagens são aplicadas em sistemas desenvolvidos na

linguagem Java. O uso da análise estática por *Rohatgi*, através do *CDG,* permite que as análises sejam feitas levando em consideração a estrutura de classes do sistema e obtendo as classes mais específicas para as características. Em *Featincode* não há a análise utilizando a estrutura de classes. Em contrapartida, em *Featincode* as matrizes permitem visualizar, não apenas as classes específicas, mas também as classes comuns entre várias características, o mesmo valendo para outros elementos do código como pacotes e métodos.

2.5 EISENBERG E VOLDER

Eisenberg e Volder (EISENBERG; DE VOLDER, 2005) mostram uma alternativa para localização de característica que usa um *ranking* heurístico (ao invés de julgamento binário de relevância) para determinar se um elemento do código é relevante para uma característica. Esta técnica é menos sensível em relação à qualidade da entrada e mais eficiente quando usado por desenvolvedores que não estão familiarizados com o sistema alvo.

Este trabalho introduz o conceito de rastro dinâmico de característica (DFT), que são artefatos que descrevem a implementação das características por ordenar elementos do código relacionados a essas características em termos da heurística (ranking gradual) que determina a relevância entre elementos do código e característica.

Um DFT compreende duas partes: 1 - o ranking, um conjunto ordenado de todos os métodos chamados durante a execução do conjunto de teste, denominado *footprint* (conjunto de elementos do código executados por um conjunto de teste); e 2 - as chamadas, um conjunto de métodos chamados entre os métodos no conjunto footprint. O conjunto de chamadas retêm informação sobre o contexto chamado para elementos particulares dentro da execução do teste. Esta informação ajuda a averiguar porque um método é relevante para uma característica.

Os DFT's são criados em três passos. Primeiro, o desenvolvedor particiona o conjunto de testes. Depois, a ferramenta DFT realiza a execução da análise de rastros e cria os conjuntos de ranking e de chamadas. A criação do DFT é automatizada depois que o mapeamento inicial teste-característica é especificado.

O ranking é um conjunto ordenado de elementos do código ordenados por quão relevantes eles são para uma característica (um método exercitado tem uma

pontuação de 0 até 1) . A pontuação do método é obtida pelo cálculo da média entre três heurísticas definidas (multiplicidade, especialização e profundidade).

A visualização dos DFTs é feita através de um navegador de código derivado de *JQuery (JANZEN; VOLDER, 2003)*. As funcionalidades de *JQuery* foram estendidas para permitir a importação de conjuntos de chamadas e *rankings* de um DFT. Três sub-navegadores permitem explorar os métodos que são ordenados de acordo com o *ranking*. O primeiro permite a navegação pelas características e os métodos associados. Dois outros navegadores permitem explorar o grafo de chamadas no sentido direto e reverso, partindo de cada método associados às características.

Comparando este trabalho com *Featincode*, percebe-se que ambos dão importância ao nível de relevância que os elementos do código possuem em relação às características. Eisenberg e Volder descrevem três heurísticas para atribuir esta relevância, enquanto *Featincode* traz matrizes com escalas de cores, mostrando quão relevante um elemento é ou não para uma característica. Os dois trabalhos utilizam análise dinâmica para a localização de características.

2.6 SINGER & KIRKHAM

O trabalho de Singer e Kirkham (SINGER; KIRKHAM, 2008) apresenta aplicações na área de visualização e *profiling* pela combinação de técnicas de atribuição de conceitos (*concept assignment*) e análise dinâmica de software (“*dynamic concept information”*). Por meio de estudos de casos, dois estilos de visualização (proporção de tempo e seqüência de execução dos conceitos) e um estudo sobre a interação entre dois conceitos são apresentados.

Segundo os autores, as informações providas pelas análises propostas são ideais para compreensão de programas, podendo ser úteis também para depuração e *profiling*. Os pontos fortes da análise estão na forma simples de instrumentação do sistema (pela inserção de comentários no código) e nas visualizações propostas que elucidam o comportamento em tempo de execução de conceitos até então inacessíveis pelo usuário (como no exemplo usado no estudo de caso com o *garbage collection*).

Em tese, os conceitos podem ser associados a qualquer nível de granularidade desde simples instruções, até grupos de classes ou interfaces.

Entretanto, a mesma simplicidade na instrumentação pode ser um dos pontos a dificultar a aplicação da técnica, pois a marcação manual dos conceitos pode ter alto custo. Além disso, para embutir os conceitos no código fonte é necessário que o analista tenha conhecimento prévio sobre os elementos relacionados a esses conceitos.

Em relação a *Featincode*, pode ser destacado que ambos combinam técnicas de análise dinâmica com "técnicas de análise semântica" (no caso, atribuição de conceitos e localização de características no código fonte) além de usar análises visuais em suas abordagens. O foco do trabalho está em duas análises: 1. tempo gasto na execução de cada conceito; 2. fases ou relação temporal entre os conceitos durante a execução. Nenhuma consideração sobre a forma de atribuição dos conceitos no código fonte é relevada. Neste trabalho o foco está justamente no mapeamento de características (que seria equivalente aos conceitos) para o código fonte. É assumido que o analista saiba previamente onde "demarcar" um conceito na implementação, enquanto em *Featincode* essa é a informação buscada. Nesse sentido os dois trabalhos poderiam ser considerados complementares. Enquanto a análise do rastro é feita sobre a proporção de tempo e da seqüência de execução de cada conceito, em *Featincode* busca-se mostrar pontos de interseção entre os elementos do código e a característica exercitada. Dessa forma, a contribuição de cada elemento do código na implementação da característica é destacada. Além disso, são mostrados os relacionamentos entre elementos do código com base nas características comuns, bem como, o relacionamento das características com base nos elementos do código em comum. Embora os autores sugiram que as técnicas poderiam ser estendidas para lidar com programas *multi-thread* e sobreposição de conceitos, isso não é tratado. Em *Featincode*, a presença de várias *threads* é considerada, bem como a existência de implementações compartilhadas por várias características.

2.7 LUKOIT & WILDE

Em (LUKOIT et al, 2000), Lukoit & Wilde apresentam a ferramenta *TraceGraph*, desenvolvida para dar suporte ao processo de localização de característica, particularmente para sistemas grandes e interativos.

TraceGraph é uma ferramenta que combina dois conceitos: *feedback* imediato

durante a execução do programa (os arquivos de rastro são escritos e analisados continuamente, enquanto o programa executa) e visualização gráfica dos resultados (semelhante a um osciloscópio que estende a visualização lentamente para a direita à medida que o programa executa).

Na visualização do *TraceGraph,* cada coluna correspondente ao período de tempo (5 segundos de execução) e cada linha a um componente do software. As células são pintadas de cinza se o componente foi executado naquele período de tempo ou branca caso contrário. Células pretas são usadas para enfatizar a primeira vez que um componente é executado.

Com *TraceGraph* um mantenedor de software será capaz de ver como o programa responde a diferentes ações. Células pretas, por exemplo, podem demonstrar um ponto interessante a ser investigado, desde que elas não apareçam até a característica ser executada. Durante todo o tempo, informações indesejadas podem se eliminadas. As diferenças entre componentes executados “com” e “sem” a característica podem ser facilmente distinguidos pelo padrão de células.

TraceGraph traz informações a respeito de “quando” determinado componente é executado durante a execução. *Featincode* mostra quando o componente é executado na seqüência de eventos do rastro e se há interseção com os eventos associados às características (intervalos de eventos no *TraceFileViewer*). As escalas de cores utilizadas em *Featincode* mostram não apenas a presença de determinado elemento na característica, mas também sua relevância baseado em diferentes métricas do rastro. Segundo os autores, *TraceGraph* compartilha algumas das limitações da técnica *Software Reconnaissance* tais como: sua aplicabilidade restrita às características controláveis pelo usuário, excluindo assim o acesso a elementos “comuns” no código; dependência dos dados de entrada, o que pode ocasionar problemas com falso positivos e com elementos não cobertos pelos em função do roteiro de execução escolhido. *Featincode* resolve a questão dos elementos comuns pelo uso das escalas de cores que graduam a participação dos elementos nas características, permitindo distinguir entre elementos mais específicos e elementos mais genéricos (ou comuns).

2.8 LIU E MARCUS

Liu e Marcus (LIU et al, 2007) apresentam uma técnica semi-automática para

localização de características no código fonte por meio de buscas textuais combinadas à análise de rastros de execução e de comentários/identificadores no código fonte. A técnica combina análise estática, dinâmica e recuperação de informações baseada em *LSI* (*Latent Semantic Index*). O principal diferencial da abordagem está no uso de um único rastro de execução que exercite a característica de interesse, sendo por isso denominada *SITIR* (*SIngle Trace and Information Retrieval*). O objetivo da abordagem é classificar os métodos dentre os dez melhores (mais relevantes para a características de interesse).

A metodologia é composta dos seguintes passos:

1. Formulação e execução de um único roteiro de execução: o sistema é executado e os pontos onde os dados (rastro) devem ser coletados são informados pelos usuário (similar a *Featincode*);
2. Formulação da consulta: o usuário formula uma consulta com base em um conjunto de termos que descrevam a característica; a ferramenta consulta os termos presentes no vocabulário do código e, caso não encontre, sugere similares ou elimina da consulta;
3. *Classificação* dos métodos executados: baseado no índice *LSI*, o conjunto de métodos gerados em 1 é ordenado com base na similaridade entre os métodos e a consulta do usuário;
4. Verificação dos resultados: o usuário inspeciona os métodos classificados no passo 3, começando pelos métodos no topo da lista. O usuário decide se cada método da lista se relaciona à característica e, em caso positivo, SITIR pára. Caso contrário o processo pode ser repetido, reformulando o roteiro, a consulta ou ambos.

A técnica LSI é utilizada para a indexação do código fonte. O índice é criado pela extração de identificadores e comentários no código fonte. Os métodos são classificados com base na sua similaridade textual entre os termos da consulta feita pelo usuário (descrevendo a característica) e os termos no índice. A indexação do código fonte do sistema é feita uma única vez e só precisa ser refeita em caso de mudanças significativas no código fonte

Os pontos fortes da abordagem estão em sua simplicidade (único rastro, marcação das características opcional (apenas para melhorar a precisão), consulta com termos descrevendo as características, resultados listados seguindo a ordem de relevância dos elementos (classes ou métodos) para as características) e

precisão (comparado a outros trabalhos, a abordagem fornece resultados tão bons ou melhores).

Os estudos de casos realizados partiram de contextos práticos (requisições de mudança e *bugs*) e foram validados com situações reais, sendo comparados com elementos efetivamente modificados pelas mudanças nessas características.

Um dos problemas potenciais é a sensibilidade dos resultados à entrada, implicando em certa dependência do conhecimento do usuário. Assim, quanto mais conhecimento o usuário tem sobre o sistema melhor serão as consultas e seus resultados. Outra questão é que a localização das características tem uma forte dependência da convenção de nomes utilizada no implementação do sistema.

Em relação a *Featincode*, a essência dos dois trabalhos é a mesma, localização de características no código. Além disso, os dois trabalhos fazem uso de de análise dinâmica com “rastro marcados” pelo usuário. Por outro lado SITIR não se não se baseia em análises visuais para o mapeamento de características, mas num esquema de classificação semi-automático segundo a relevância dos elementos do código para a característica. A abordagem é híbrida, combinando informações estáticas (identificadores/comentários), informações dinâmicas (rastros de execução marcados) e informações semânticas (com base no índice criado com *RI LSI)*. Ao contrário de *Featincode*, SITIR não faz análises de similaridade entre elementos do código ou entre características.

2.9 GREEVY E DUCASSE

Greevy e Ducasse (GREEVY; DUCASSE, 2005) tratam o problema de localização de característica sobre duas perspectivas complementares: a perspectiva das características e a perspectiva das unidades computacionais. Na perspectiva das características, os autores caracterizam a participação de elementos do código em características, de forma isolada ou em grupo. Os elementos são classificados em: elementos não cobertos pela análise, elementos específicos de uma característica, elementos compartilhados por um grupo de características e elementos compartilhados por todas as características. Na perspectiva de unidades computacionais, a existência de características em elementos de código é caracterizada indicando o nível de relação entre características a respeito dos lementos compartilhados.

A base para a definição das relações está na extração de várias métricas dos rastros de execução. Algumas métricas apresentadas na abordagem *Featincode* são semelhantes às do trabalho de Greevy e Ducasse. Por outro lado, o tratamento de *múltiplas* threads e a possibilidade de explorar simultaneamente múltiplos níveis da granularidade de elementos de código em *Featincode* não são tratados do mesmo modo. *Featincode* permite alternar rapidamente entre uma visão de alto-nível do sistema para uma visão detalhada. Outra vantagem de *Featincode* é o uso de abstrações conhecidas da linguagem de programação, tal como, pacotes, classes e métodos.

Posteriormente, Greevy (GREEVY, 2007) compila várias técnicas e trabalhos na análise dinâmica focada em características. O objetivo do trabalho é acrescentar valor para as técnicas de engenharia reversa com conhecimento semântico fornecido por análise de características. Algumas contribuições principais incluem a formalização de um meta-modelo para o comportamento da característica em tempo de execução e várias análises centradas em características: análise do comportamento da característica em tempo de execução com respectivo mapeamento para elementos do código, adição de contexto da característica para visões estáticas e estruturais de elementos do código-fonte, e identificação de relações de dependência entre características. O seu trabalho também apresenta a definição da taxonomia para classificação de elementos do código segundo a sua relevância para uma ou várias características, batizada como medida de afinidade (*Affinity Measure*). Featincode usa a idéia de matrizes de visualização e escalas de cores para caracterizar a relação entre característica e código. Featincode não faz considerações específicas sobre a evolução de sistema.

2.10 Eisenbarth e Koschke

Eisenbarth e Koschke (EISENBARTH; KOSCHKE; SIMON, 2003) combinam técnicas de análises dinâmica e estática com a Análise de Formal de Conceitos para localizar as características e explorar as relações entre conjuntos de características e conjuntos de elementos do código-fonte (unidades computacionais). O processo consiste em:

1. Criação de roteiros baseados em características, que são inicialmente conhecidas ou incrementalmente identificadas por especialistas no

domínio;

2. Extração de gráfico de dependência estática entre unidades computacionais;
3. Execução do sistema usando os roteiros criados e análise dinâmica para gerar um reticulado de conceitos, os quais informam as unidades computacionais que contribuem para um dado conjunto de características.
4. Interpretação deste conceito para identificar unidades computacionais relevantes para as características selecionadas.

Neste trabalho, a análise concentra-se na análise do reticulado de conceitos, e em *Featincode* é proposta a combinação de diversas visões simultâneas envolvendo a estrutura do código fonte, o modelo de características, e o relacionamento esses elementos.

CAPÍTULO 7. CONCLUSÃO

"Embora ninguém possa voltar atrás e fazer um novo começo,
qualquer um pode começar agora e fazer um novo fim."
Chico Xavier

Neste trabalho foi mostrado como o uso de uma ferramenta de visualização pode fornecer informações úteis sobre o espalhamento de características num sistema orientado a objetos. A abordagem e a ferramenta propostas têm o intuito de ajudar desenvolvedores a obter uma melhor compreensão do sistema em tarefas de manutenção, que de outra maneira seria improvável pela simples análise do código e documentação.

As características da ferramenta *Featincode* foram descritas. Alguns detalhes sobre sua arquitetura e implementação também foram discutidos. Para auxiliar em sua aplicação uma metodologia foi definida. Com base nessa metodologia, um estudo sobre uma aplicação interativa de grande porte, ArgoUML, foi conduzido. O estudo mostrou resultados positivos quanto à capacidade da ferramenta em apontar a relação entre características e elementos do código fonte. Confirmando com a documentação, os elementos apontados pela ferramenta com maior relevância de fato mostraram-se bons pontos de partida para uma análise mais detalhada do código fonte.

A ferramenta *Featincode* ajudou a identificar e entender várias threads iniciadas pelo ArgoUML. Além disso, foi possível identificar diferentes grupos de elementos do código (pacotes, classes ou métodos) com relação às características: elementos específicos, elementos gerais, elementos com alta participação na execução e elementos não participantes nas características. Também foi possível encontrar facilmente informações sobre a intersecção em diferentes níveis de detalhamento, tanto entre elementos do código quanto entre características.

As possíveis aplicações da ferramenta no âmbito dos processos de desenvolvimento incluem a compreensão, manutenção e evolução de um sistema.

Para a compreensão um desenvolvedor poderia partir de um conjunto de características de interesse e realizar execuções do sistema com o intuito de

identificar os elementos do código relacionados a essas características. O esquema de coloração baseado em métricas oferecido permite priorizar os elementos a serem estudados seja buscando elementos mais gerais, seja buscando elementos específicos de uma dada característica. Isso pode contribuir para reduzir consideravelmente o esforço para a análise. Nesse caso, as métricas de cobertura de elementos do código e/ou características poderiam ser de maior interesse.

Partindo para a manutenção, o desenvolvedor poderia fazer o uso da ferramenta com o intuito de identificar os pontos onde as mudanças no sistema devem ser aplicadas. O desenvolvedor poderia especificar roteiros de execução do sistema contendo características relacionadas às atividades de manutenção. Outra alternativa para apoiar a manutenção consiste no reconhecimento da modularidade e coesão do sistema quanto às características. As matrizes de similaridade seriam ideais para esse tipo de análise.

Finalmente, para atividades de evolução o desenvolvedor poderia partir de características similares já implementadas no sistema e que tenham relação com as evoluções a serem introduzidas e utilizar a ferramenta para buscar elementos do código em comum e essenciais entre essas características, bem como elementos específicos de cada uma delas. Nesse caso, as matrizes Elemento x Elemento e Característica x Característica poderiam mais adequadas.

Algumas perguntas ainda permanecem para trabalhos futuros. Outros experimentos precisam ser executados com novos sistemas e/ou variando o conjunto de características selecionadas. Seguindo os resultados em (SANGAL et al, 2005), espera-se que o uso de algoritmos de reorganização hierárquica possa sugerir possíveis refatorações que realcem a modularidade acerca das características implementadas e, conseqüentemente, possibilite reduzir o espalhamento dessas características. Ainda, pela filtragem e reconhecimento de padrões no rastro de execução (HAMOU-LHADJ; LETHBRIDGE, 2002)(HAMOU-LHADJ; LETHBRIDGE, 2005)(HAMOU-LHADJ; LETHBRIDGE, 2006), é esperado que a influência de eventos desinteressantes causados por laços, chamadas recursivas e seqüências redundantes seja removida. A análise de escalabilidade da abordagem e da ferramenta também devem ser consideradas.

Como melhorias na ferramenta, ficam como sugestões para implementações posteriores: a adição de outros esquemas de filtragem (por exemplo, com base nas escalas de cores), adição e teste de novas métricas, melhoria na performance da

indexação e cálculos de métricas, suporte para o tratamento de múltiplos rastros de execução, detecção de problemas com a ordem de execução das características, integração com ambientes de desenvolvimento (por exemplo, gerar uma versão de Featincode como um plugin Eclipse), integração com ferramentas de modelagem de características.

Outra melhoria seria a adaptação da ferramenta para distinguir entre rastros brutos e rastros comprimidos, tirando proveito de informações adicionais que o rastro comprimido possa oferecer sobre os pontos onde houve compressão. O mesmo pode ser dito para rastros que tenham passado por uma compressão baseada em sumarização (HAMOU-LHADJ; LETHBRIDGE, 2006).

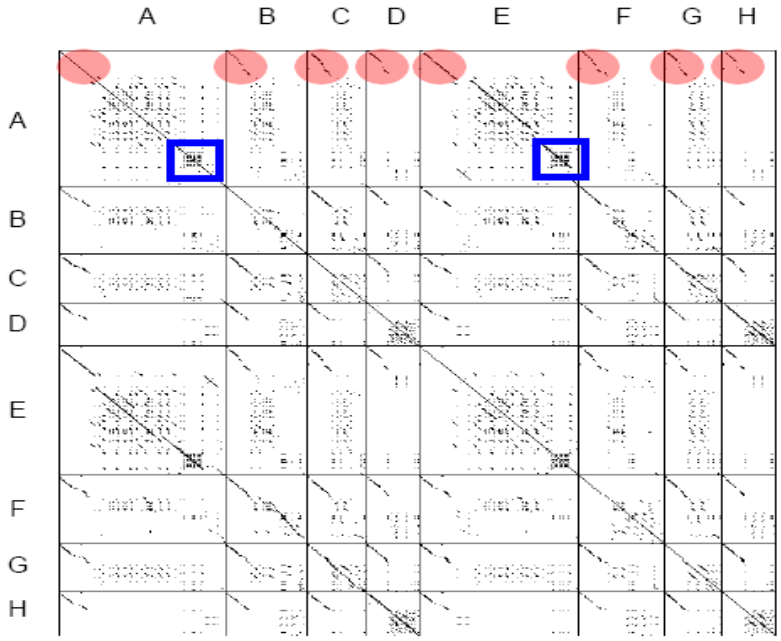


Figura 7.1: Aplicação da técnica *Dotplot* a uma seqüência de eventos do rastro de execução (apenas ilustrativa, não correspondendo a dados reais).

Uma alternativa que poderia casar bem com a técnica *Dotplot* e conseqüentemente permitir o reconhecimento de um volume maior de padrões é a utilização dos eventos do rastro como base para as matrizes. A adaptação do sistema para considerar a seqüência de eventos é uma alternativa promissora para a análise do rastro, podendo facilitar a filtragem de eventos e a identificação de seqüências recorrentes. Adicionalmente, as características poderiam ser mapeadas para as seqüências de forma a facilitar a identificação da semântica desses padrões. A Figura 7.1 ilustra a idéia. As letras “A” a “H” poderiam corresponder a passos de

execução de uma característica. As células marcadas com um círculo poderiam corresponder a uma seqüência de eventos que ocorre inicialmente em “A” e que se repete ao longo de todos os demais passos da execução. As células marcadas com um quadrado seriam equivalentes a um conjunto de métodos chamados repetidamente durante o passo de execução “A” e que voltam a acontecer no passo “E”, por exemplo, em função de uma chamada recursiva ou laço de execução. Essas são algumas das possibilidades para a identificação de padrões em um rastro de execução contendo milhares de eventos por meio de uma simples interpretação visual.

Até o momento em que a versão final deste texto estava sendo escrita, alguns artigos foram aceitos para a publicação em conferências (SOBREIRA; MAIA, 2008a, 2008b, 2008c) e um periódico (SOBREIRA; MAIA, 2008d). Estes trabalhos foram derivados a partir dos resultados desta dissertação.

Referências Bibliográficas

ANTKIEWICZ, M.; CZARNECKI, K. FeaturePlugin: feature modeling plug-in for Eclipse, In: PROCEEDINGS OF THE 2004 OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE (ECLIPSE'04), New York, NY, USA, 2004, p.67-72.

ANTONIOL, G.; GUÉHÉNEUC, Y-G. Feature Identification: A Novel Approach and a Case Study, In: PROCEEDINGS OF THE 21ST IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM '05), Washington, DC, USA, 2005, p.357-366.

ANTONIOL, G.; GUÉHÉNEUC, Y-G. Feature Identification: An Epidemiological Metaphor, **IEEE Transactions on Software Engineering**, v.32, n.9, p.627-641, 2006.

APACHE SOFTWARE FOUNDATION - ASF, **Apache Logging Services**, Disponível em: <http://logging.apache.org>. Acesso em: 25 ago. 2008.

ASPECTC.ORG, **AspectC++**, Disponível em: <http://www.aspectc.org>, Acesso em: 25 ago. 2008.

BALL, T., The concept of dynamic analysis, In: PROCEEDINGS OF THE 7TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH THE 7th ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE-7), London, UK, 1999, 216-234.

BEDNASCH, T.; LANG, M. **Captain Feature 1.0**, Disponível em: <https://sourceforge.net/projects/captainfeature>, Acesso em: 08 ago. 2008.

BIGGERSTAFF, T. et al. Program understanding and the concept assignment problem, **Comm. ACM**, v.37, n.5, p.72-82, 1994.

BONE, J.; VASSEUR, A. **AspectWerkz**, Disponível em: <http://aspectwerkz.codehaus.org>, Acesso em: 25 ago. 2008.

CABRAL, Facundo. **No estas deprimido**, Disponível em: <http://www.facundocabral.org/index.php?ID=66>. Acesso em: 20/09/2008.

CHEN, Tie-Qi, **LTFViewer - Large Text File Viewer 4.1**, Disponível em: <http://www.swiftgear.com>, Acesso em: 26 ago. 2008, 2008.

CHURCH; HELFMAN, Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code, **Journal of Computational and Graphical Statistics**, v.2, n.2, p.153-174, 1993.

CORNELISSEN, B. et al. Understanding execution traces using massive sequence and circular bundle views, In: PROC. OF THE 15th INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC'07), 2007, p.49-58.

CZARNECKI, K.; EISENECKER, U. **Generative Programming: Methods, Techniques and Applications**, Addison Wesley, 2000.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. W. Staged Configuration Using Feature Models, In: PROCEEDINGS OF THE 3rd SOFTWARE PRODUCT-LINE CONFERENCE (SPLC'04), 2004, p.266-283.

EADDY, M. et al. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis, In: 16th IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC'08), Amsterdam, The Netherlands, 2008.

EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Feature-Driven Program Understanding Using Concept Analysis of Execution Traces, In: PROCEEDINGS OF THE 9th INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION (IWPC '01), Washington, DC, USA, 2001, p.300.

EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating Features in Source Code, **IEEE Transactions on Software Engineering**, v.29, n.3, p.210-224, 2003.

EISENBERG, A. D.; DE VOLDER, K., Dynamic Feature Traces: Finding Features in Unfamiliar Code, In: PROCEEDINGS OF THE 21st IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'05), Washington, DC, USA, 2005, p.337-343.

FIERZ, J. **Wiretap - Feature and Object Flow Analysis for Java**, Disponível em: <http://smallwiki.unibe.ch/wiretap-featureandobjectflowanalysisforjava>, Acesso em: 25 ago. 2008.

FORGET, A.; ARNOLD, D.; CHIASSON, S. CASE-FX: Feature Modeling Support in an OO Case Tool, In: COMPANION TO THE 22nd ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS (OOPSLA

'07), New York, NY, USA, 2007, p.803-804.

GAMMA et al. **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison Wesley Longman, 1995.

GEAR, A. L. et al. Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling, In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, 2005, p.34-43.

GREEVY, O., DUCASSE, S. Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces, In: PROCEEDINGS OF 6th INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REENGINEERING (WOOR'05), 2005b.

GREEVY, O.; DUCASSE, S.; Correlating Features and code using a compact two-sided trace analysis approach, In: PROC. OF THE 9th EUROPEAN CONF. ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR'05), 2005, p.314-323.

GREEVY, Orla. **Enriching Reverse Engineering with Feature Analysis**, Phd. Thesis, Berne University, Switzerland, 2007.

GRUBER, H.; JORDAN, J. **Tjger - The Java Game Engine to Reuse**, Disponível: <http://sourceforge.net/projects/tjger>, Acesso em: 24 jun. 2008.

HAMOU-LHADJ, A.; LETHBRIDGE, T.C. A Metamodel for Dynamic Information Generated from Object-Oriented Systems, **Electronic Notes Theoretical Computer Science**, v.4, p.59-69, 2004.

HAMOU-LHADJ, A.; LETHBRIDGE, T.C. Compression Techniques to Simplify the Analysis of Large Execution Traces, In: PROC. OF THE 10th IEEE INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION (IWPC), Paris, France, 2002, p.159-168.

HAMOU-LHADJ, A.; LETHBRIDGE, T.C. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools, In: PROCEEDINGS OF THE 10th IEEE INTERNATIONAL CONFERENCE ON ENGINEERING OF COMPLEX COMPUTER SYSTEMS (ICECCS'05), Washington, DC, USA, 2005, p.559-568.

HAMOU-LHADJ, A.; LETHBRIDGE, T.C. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system, In: PROC. OF THE 14th. INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), 2006, p.181-189.

HELFMAN, Dotplot Patterns: A Literal Look at Pattern Languages, **Theory and Practice of Object Systems**, v.2, n.1, p.31-41, 1996.

HELFMAN, Helfman, Similarity Patterns in Language, In: IEEE SYMPOSIUM ON VISUAL LANGUAGES, 1994, p.173-175.

IBRAHIM et al. Case study: Reconnaissance techniques to support feature location using RECON2, In: 10th ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC'03), 2003, p.371.

JANZEN, D.; VOLDER, K. Navigating and querying code without getting lost, In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD'03), New York, NY, USA, 2003, p.178-187.

JARZABEK, S. **Effective Software Maintenance and Evolution: A Reuse-Based Approach**, Auerbach Publications, 2007.

JIANG, M. et al. Software Feature Understanding in an Industrial Setting,In: PROCEEDINGS OF THE 22nd IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'06), Washington, DC, USA, 2006, p.66-67.

JIANG, M. et al. **TraceGraph 4:** A Demonstration Case Study, Technical Report SERC-TR-290, Software Engineering Research Center, 2007.

KANG et al. **Feature-Oriented Domain Analysis (FODA): Feasibility Study**, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University,1990.

KO, A. et al. Information Needs in Collocated Software Development Teams, In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE'07), Washington, DC, USA, 2007, p.344-353.

KO, A. J. et al. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, **IEEE Transactions on Software Engineering**, v.32, n.12, p.971-987, 2006.

LEHMAN, M. M. et al. Metrics and Laws of Software Evolution - The Nineties View, In: PROCEEDINGS OF THE 4th INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS (METRICS'97), Washington, DC, USA, 1997, p.20.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution, In: PROC.

IEEE (SPECIAL ISSUE ON SOFTWARE ENGINEERING), 1980, p.1060-1076.

LIU, D. et al. Feature location via information retrieval based filtering of a single scenario execution trace, In: PROCEEDINGS OF THE 22nd IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'07), New York, NY, USA, 2007, p.234-243.

LUKOIT et al. **TraceGraph: Immediate Visual Location of Software Features**, Technical Report SERC-TR-86-F, Software Engineering Research Center, Purdue University, 2000.

MURPHY, G. C; NOTKIN, D.; SULLIVAN, K. J. Software reflexion models: bridging the gap between design and implementation, **IEEE Transactions on Software Engineering**, v.27, n.4. 2001.

POSHYVANYK et al. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval, **IEEE Transactions on Software Engineering**, v.33, n.6, p.420-432, 2007.

RAJLICH, V.; WILDE, N. The role of concepts in program comprehension, In: **Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02)**, 2002, p.271-278.

ROHATGI, A.; HAMOU-LHADJ, A.; RILLING, J. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis, In: 16th IEEE INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), Amsterdam, The Netherlands, 2008.

ROHLIK; O. PASETTI, A. Ondrej Rohlik and Alessandro Pasetti, **XFeature - Feature Modeling Tool**, Disponível em: <http://www.pnp-software.com/XFeature>, Acesso em: 08 ago. 2008.

SALAH, M. et al. Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems, In: PROCEEDINGS OF THE CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR'06), Washington, DC, USA, 2006, p.71-80.

SANGAL, N. et al. Using dependency models to manage complex software architecture, In: PROCEEDINGS OF THE 20th ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA'05), 2005, p.167-176.

SAUER, F. **Metrics 1.3.6**, Disponível em: <http://metrics.sourceforge.net>. Acesso

em: 02 ago. 2008.

SEI, **Framework for Software Product Line Practice Version 5.0**, Disponível em: <http://www.sei.cmu.edu/productlines/framework.html>, Acesso em: 05 ago. 2008, 2008a.

SEI, **Model Based Software Engineering**, Disponível em: <http://www.sei.cmu.edu/mbse>, Acesso em: 05 ago. 2008, 2008b

SEI, **Software Engineering Institute**, Disponível em: <http://www.sei.cmu.edu., Acessado em: 05 ago. 2008, 2008c

SINGER, J.; KIRKHAM, C. Dynamic analysis of Java program concepts for visualization and profiling, **Sci. Comput. Program.**, v.70, n.2-3, p.111-126, 2008.

SOBREIRA, V.; MAIA, M. A. Análise do Espalhamento de Características pela Interpretação Visual de Rastros de Execução, In: ANAIS DO 5º WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA (WMSWM'08), Florianópolis-SC, Brasil, Junho, 2008a.

SOBREIRA, V.; MAIA, M. A. Featincode: Suporte à Compreensão do Espalhamento de Características pela Interpretação Visual de Rastros de Execução, In: ANAIS DO 22º SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE - SESSÃO DE FERRAMENTAS (SBES'08), Campinas-SP, Brasil, Outubro, 2008b.

SOBREIRA, V.; MAIA, M. A. A Visual Trace Analysis Tool for Understanding Feature Scattering, In: PROCEEDINGS OF THE 15th WORKING CONFERENCE ON REVERSE ENGINEERING - TOOL DEMO (WCRE'08), Antwerp, Bélgica, Outubro, 2008c.

SOBREIRA, V.; MAIA, M. A. Analyzing Feature Scattering with Visual Information of Execution Traces, **Infocomp - Journal of Computer Science**, Accepted: 15 set. 2008, (Aguardando publicação), 2008d.

SUN MICROSYSTEMS, **JVMTM Tool Interface (JVM TI)**, Disponível: <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti>, Acesso em: 25 ago. 2008, 2008.

THE ECLIPSE FOUNDATION, **AspectJ**, Disponível em: <http://www.eclipse.org/aspectj>, Acesso em: 25 ago. 2008, 2008.

TIGRIS.ORG, **ArgoUML**, Disponível em: <http://argouml.tigris.org>, Acesso em: 24 jun. 2008, 2008.

WIKIPEDIA, **Programação orientada a aspectos**, Disponível: <http://pt.wikipedia.org/wiki/Programação_orientada_a_aspecto>, Acesso em: 25 ago. 2008.

WILDE, N. et al, A comparison of methods for locating features in legacy software, **Journal of Systems and Software**, v.65, n.2, p.105-114, 2003.

WILDE, N. **Faster Reuse and Maintenance Using Software Reconnaissance**, Technical Report SERC-TR-75-F, Software Engineering Research Center, University of Florida,1994.

WILDE, N. **RECON Tools for Software Engineers**, Disponível em: <http://www.cs.uwf.edu/~recon>, Acesso em: 11 ago. 2008,

WILDE, N.; CASEY, C. Early field experience with the Software Reconnaissance technique for program comprehension, In: PROCEEDINGS OF THE 3rd WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE '96), Washington, DC, USA, 1996, p.270.

WILDE, N.; SCULLY, M. C. Software Reconnaissance: Mapping Program Features to Code, **Journal of Software Maintenance: Research and Practice**, v.7, n.1, p.49-62, 1995.

WONG, E. et al, Locating program features using execution slices, In: PROCEEDINGS OF THE 2nd IEEE SYMPOSIUM ON APPLICATION-SPECIFIC SYSTEMS AND SOFTWARE ENGINEERING TECHNOLOGY, 1999, p.194--203.

ZHAO, W. et al, SNIAFL: Towards a Static Non-Interactive Approach to Feature Location, In: PROCEEDINGS OF THE 26th INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE'04), Washington, DC, USA, 2004, p.293-303.