Análise do Espalhamento de Características pela Interpretação Visual de Rastros de Execução

Victor Sobreira, Marcelo de Almeida Maia

Departamento de Ciência da Computação
Universidade Federal de Uberlândia (UFU) – Uberlândia – MG – Brasil

victor.sobreira@gmail.com, marcmaia@facom.ufu.br

Abstract. *Feature location in the source code still is a challenge, specially when the source-code is not modularized in a feature-fashioned way, causing feature scattering. This work proposes a method for understanding feature scattering through the graphical interpretation of relationships between feature elements and source code elements. The proposed tool collects and represents trace events of multithreaded programs and shows some graphics that help in feature scattering analysis. The proposed method and tool are validated with the analysis of some features of the ArgoUML CASE tool. The conclusion is that interesting information about feature location can be straightforwardly found.*

Resumo. *A localização de características no código-fonte ainda é um desafio, especialmente quando o código não foi modularizado com base em características, causando assim o espalhamento. Este trabalho propõe um método para o entendimento do espalhamento de características pela interpretação visual de relacionamentos entre características e elementos do código fonte. A ferramenta proposta coleta e representa rastros de execução de programas multi-thread e mostra alguns gráficos que auxiliam na análise do espalhamento. O método proposto e a ferramenta são validados com a análise de algumas características da ferramenta CASE ArgoUML. A conclusão é que informações interessantes sobre a localização das características podem ser encontradas de forma simples.*

1. Introdução

As melhores práticas de desenvolvimento orientado por objetos têm contribuído para muitos fatores de qualidade interna do software. Duplicações tendem a ser minimizadas por construções polimórficas das linguagens de programação e pela aplicação contínua de padrões de projeto. Este é um benefício que se estende também para a produtividade e manutenção. Porém, o entendimento do código-fonte sob a perspectiva do usuário ainda não dispõe de um mapeamento direto dos requisitos para elementos do código.

A rastreabilidade entre requisitos e artefatos de código ainda é um grande desafio para desenvolvedores, pois uma simples interação do usuário pode disparar o uso de centenas, talvez milhares de classes e métodos em um sistema. Este cenário impõe dificuldades para o entendimento do sistema. Uma opção é depurar e navegar pelo código usando facilidades de ambientes de desenvolvimento. Entretanto, mesmo em sistemas de porte médio, esta tarefa é normalmente muito custosa pela grande quantidade de chamadas de métodos envolvidas. Outra solução é analisar os rastros de execução derivados do evento. De fato, já foram propostos muitos trabalhos para auxiliar essa análise baseados na informação que se deseja extrair [1,2,3,4,5,6,7,8].

A nossa hipótese é que a capacidade de descobrir informações interessantes sobre a implementação de características pode ser aprimorada se permitimos aos desenvolvedores navegar por matrizes relacionando código-fonte e características.

Neste trabalho propomos o uso de uma ferramenta visual para analisar a relação entre características e elementos do código fonte sob diferentes perspectivas, expressas em matrizes de visualização. A principal contribuição deste artigo é mostrar como as matrizes de visualização podem prover um discernimento útil para o entendimento do espalhamento das características pelo código de sistemas de grande porte.

O artigo é organizado como se segue. A Seção 2 apresenta um resumo sobre a ferramenta de visualização proposta, resume a forma de coleta e representação do rastro e mostra a definição das matrizes de visualização propostas. A Seção 3 relata o estudo de caso realizado. A Seção 4 apresenta a discussão sobre os resultados encontrados. A Seção 5 apresenta trabalhos relacionados. A última seção apresenta a conclusão e trabalhos futuros.

2. Visão Geral e a Ferramenta de Visualização

O objetivo de nossa abordagem é entender como as características se espalham no código fonte. Consideramos como característica *"uma unidade observável de comportamento do sistema disparada pelo usuário"*[9].

Primeiramente, o desenvolvedor define quais cenários de execução são interessantes para a análise e à manutenção a ser realizada. Para isso a consulta à documentação ou a usuários do sistema pode ser necessária. Esses cenários devem ser expressos em termos de características. O sistema alvo deve ser recompilado com suporte a aspectos para a coleta do rastro por nossa ferramenta de instrumentação, baseada em AspectJ.

Imediatamente antes da execução de um cenário, deve se informar à ferramenta de instrumentação que os eventos (chamadas de métodos) a partir daquele ponto estão relacionados a uma nova característica. Similarmente, ao fim de todos os passos necessários para a execução da característica a ferramenta deve ser notificada.

Dois tipos de arquivos são gerados: 1. arquivos com o rastro da execução para cada *thread* iniciada pelo sistema alvo; 2. arquivo de marcação dos instantes de início e fim de cada característica executada.

Os arquivos são numerados na mesma ordem de início das *threads*. Para cada chamada, uma linha é gerada no arquivo da *thread* correspondente. A linha registra: o *nome qualificado do método*, incluindo o *pacote* e o nome da *classe*, e uma *marca de tempo,* indicando o instante de execução da entrada no método.

A ferramenta de visualização lê os rastros gerados e abre as janelas mostradas na Figura 1. 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 cenários de execução. As características podem ser organizadas hierarquicamente, devendo tal organização ser informada em um arquivo externo. A janela *C* mostra as *threads* que foram iniciadas durante a coleta do rastro e os percentuais de participação dos elementos no rastro. A janela *D* mostra matrizes de intersecção 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 mais detalhes da métrica mostrada na janela *E*. Neste exemplo, as seis características comuns entre o pacote `org.argouml.model` e o pacote `org.tigris.gef.graph` são mostradas.

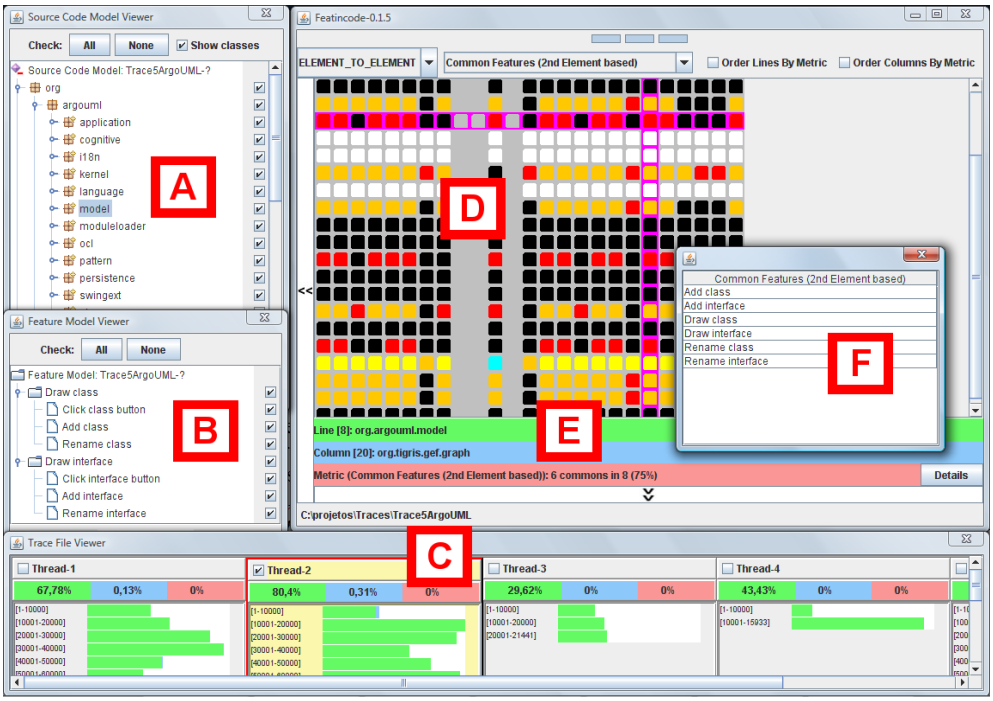

Figura 1: Resumo da ferramenta de Visualização

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 marcas delimitando o início e fim da execução de cada característica. Os relacionamentos são calculados para cada *thread*, permitindo ao analista focar sua atenção em *threads* específicas ou mesmo comparar os eventos entre elas.

Quadro 1: Detalhes das matrizes calculadas.

	Objetivo	***Métricas***
Elemento x Elemento	Mostra características comuns entre os elementos do código com o intuito de representar a similaridade de implementação entre pacotes, classes e métodos	1. $\|Fr \cap Fc\| / \|Fr\|$ 2. $\|Fr \cap Fc\| / \|Fc\|$ 3. $\|Fr \cap Fc\| / \|F\|$
Elemento x Característica	Mostra a influência que um elemento do código tem na implementação da característica, e inversamente, quais características um elemento do código ajuda a implementar	1. $\|Le \cap Lf\| / \|Lf\|$ 2. $\|Le \cap Lf\| / \|Le\|$
Característica x Característica	Mostra elementos comuns entre as características visando expressar uma noção de similaridade entre elas e o espalhamento sobre os elementos código fonte presentes no rastro.	1. $\|Er \cap Ec\| / \|Er\|$ 2. $\|Er \cap Ec\| / \|Ec\|$ 3. $\|Er \cap Ec\| / \|E\|$
Fr e ***Fc:*** conjunto de características associadas aos elementos na linha ***r*** e na coluna ***c***; ***F:*** conjunto de todas as características demarcadas no rastro durante a execução do sistema; ***Le*** e ***Lf***: conjunto de linhas no rastro onde o elemento do código ***e*** e a característica ***f***, ocorrem; ***Er*** e ***Ec***: conjunto de elementos do código associados às características na linha ***r*** e coluna ***c;*** ***E***: o conjunto de todos os elementos do código presentes no rastro.		

As características e os elementos do código são indexados usando as informações de cada evento registrado. Os índices especificam os intervalos de execução para cada elemento no rastro.

Algoritmos de intersecção são aplicados sobre esses intervalos para definir o relacionamento entre os elementos. A indexação permite a geração das matrizes em tempo de execução com um tempo de resposta razoável para o usuário. A indexação permite a geração das matrizes em tempo de execução com um tempo de resposta razoável para o usuário. Com base no valor da métrica calculada, cada célula da matriz é colorida de acordo com o padrão de cores mostrado na Figura 2. Um dos objetivos do uso dessa escala é o de facilitar a caracterização de

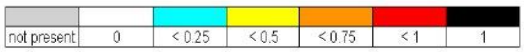

Figura 2: Escala de cores e seus respectivos intervalos.

elementos com métricas similares. O Quadro 1 mostra os detalhes para cada uma das matrizes.

3. Estudo de Caso

Um estudo de caso foi realizado para compreender o espalhamento de algumas características da ferramenta CASE ArgoUml (http://argouml.tigris.org). Para esta análise foram escolhidas duas características básicas numa ferramenta CASE UML, mostradas na janela B da Figura 1: *Draw class* e *Draw interface*. A seqüência de eventos foi: *Click class button, Add class, Rename class, Click interface button, Add interface* e *Rename interface*. Essas características poderiam ser alvo de uma possível manutenção na ferramenta e, sendo assim, seria importante entender como elas se espalham pelo código fonte.

3.1. Analisando a semântica das *Threads*

Cinco *threads* foram iniciadas (Figura 1). As *threads Thread-1* e *Thread-4* foram descartadas, pois estavam associadas exclusivamente à inicialização do sistema. Isso pôde ser constatado pelo fato de que as células de todas as matrizes destas *threads* foram renderizadas na cor cinza. As *threads Thread-2* e *Thread-5* possuíam o maior número de chamadas associadas às características selecionadas. As matrizes *Elemento x Característica* dessas *threads* são mostradas na Figura 3. Em uma primeira análise constatamos que a matriz da Thread-5 tem mais linhas homogêneas do que a matriz da Thread-2. Isso significa que a Thread-5 é menos influenciada pelas características exercitadas. Uma análise mais aprofundada da matriz da Thread-5 foi feita pela expansão de nós dos elementos do código correspondentes às células não-brancas e não-cinzas na matriz, obtendo uma segunda informação interessante mostrada na Figura 4: a maioria dos elementos expandidos tem duas características em comum: 1. eles pertencem ao pacote `cognitive` e seus sub-pacotes, em especial, `critics`; 2. as classes correspondentes tem um prefixo `Cr` indicando que são responsáveis pela geração de críticas. Essa informação é coerente, pois como as críticas são geradas em *background*, elas são menos influenciadas pela característica em execução. Assim, podemos ver que a *Thread-5* é responsável por gerar críticas sobre o modelo desenhado. Através de uma análise similar, constatamos que a *Thread-3* é similar à *Thread-5*, sendo também responsável por gerar críticas. Por exclusão, a *Thread-2* é responsável pela interação com o usuário.

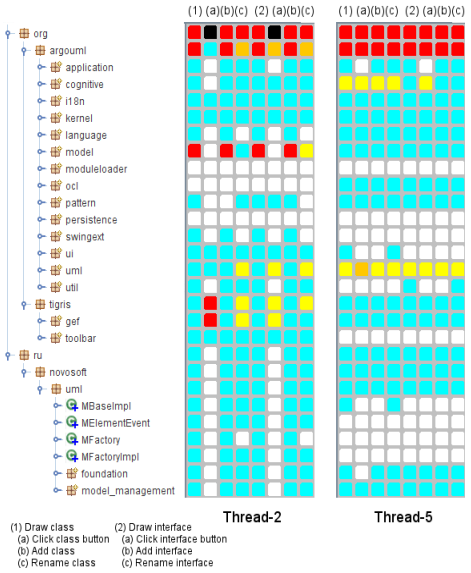

Figura 3: Elementos do código participantes nas características das Threads 2 e 5.

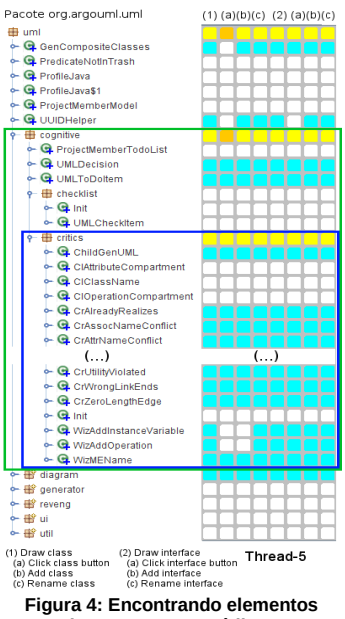

Figura 4: Encontrando elementos importantes no código.

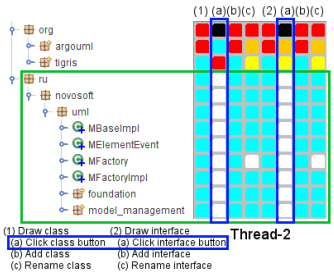

Figura 5: Participação do pacote ru nos cliques do mouse.

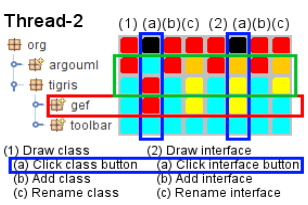

Figura 6: Detecção de uma discrepância aparente entre características similares.

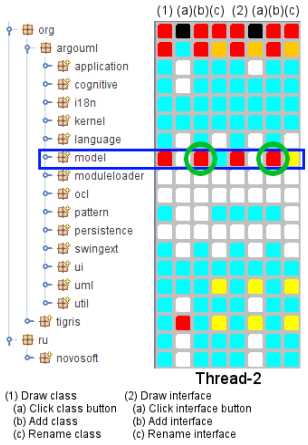

Figura 7: Encontrando elementos importantes do código.

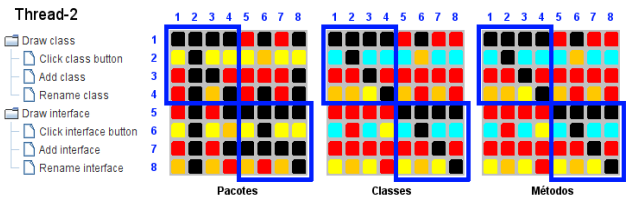

Figura 3: Matriz Característica x Característica.

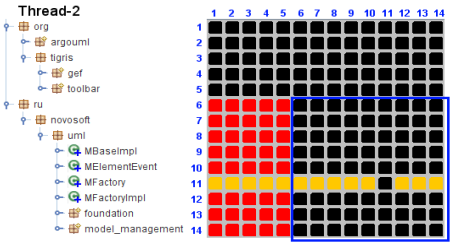

Figura 9: Matriz Elemento x Elemento.

3.2. Analisando a participação de Elementos do Código nas Características

Considerando a maior influência da *Thread-2* sobre as características selecionadas, usaremos as matrizes dessa *thread* para analisar a influência de pacotes de mais alto nível em cada característica. A Figura 5 mostra uma matriz *Elemento x Característica* onde destacamos os pacotes `org` e `ru`, e as características *Click class button* e *Click interface button*. Uma análise direta é que o pacote `ru` não participa em cliques de botão em nenhum dos dois casos (desenho de classes e de interfaces). O pacote `ru` contém a API NSUML, responsável pelo meta-modelo de UML na implementação do ArgoUML. Isto é uma descoberta coerente, pois um evento de interface que não manipula o modelo de UML desenhado não deveria necessitar de nenhuma funcionalidade do pacote de meta-modelo.

Outro ponto de destaque na mesma matriz é a diferença entre a implementação de características similares tais como *Click class button* e *Click interface button.* Como mostrado na Figura 6, os pacotes `org.argouml` e `org.tigris` tem diferentes níveis de participação nessas características. Após a expansão dos pacotes, podemos encontrar quais sub-pacotes são responsáveis pela discrepância (`org.tigris.gef` e `subpacotes`). Em uma análise direta, observamos que quando executamos a característica *Click interface button*, a sub-árvore para `org.argouml.uml.diagram.static_structure.ui` chama métodos das classes `FigClass` e `SelectionClass`. A primeira classe é responsável por desenhar uma classe na tela e a segunda por gerenciar a sua seleção. Esta situação, à primeira vista, parece ser incoerente porque a característica está relacionada ao conceito de interface e não ao conceito de classe. Realizamos uma nova execução, dessa vez com *Draw interface* antes de *Draw class* para verificar se a ordem de execução tem alguma influência. O resultado foi semelhante, porém agora os métodos das classes `FigInterface` e `SelectionInterface` foram chamados. A razão foi que o diagrama só é atualizado após um botão ser clicado e, portanto o elemento desenhado antes (classe ou interface) tem os respectivos métodos chamados.

Para as características *Add class* e *Add interface*, na 3ª e 7ª coluna da matriz respectivamente, um fato esperado ocorreu (Figura 7). Elas são semelhantes e a maior parte das chamadas (> 75 %, de acordo com a escala de cores) está localizada no pacote `org.argouml.model`. Isto é uma informação importante pois indica que o nível de espalhamento desta característica é baixo, e, como conseqüência, a compreensão detalhada deste pacote é essencial para entender a implementação da característica.

3.3. Analisando pontos comuns entre as Características

A Figura 8 mostra as matrizes possíveis para as características *Draw class* e *Draw interface*. As

diagonais principais são todas pretas, pois se trata da mesma característica na linha e na coluna. As primeiras colunas em cada quadrado marcado são pretas. Essas linhas correspondem às super-características *Draw class* e *Draw interface.* Essas características representam a composição das sub-características (*Click class/interface button*, *Add class/interface* e *Rename class/interface*) onde todos os elementos do código que participam em uma sub-característica necessariamente participam da super-característica. As diagonais dos quadrados não marcados são onde ocorre o casamento das mesmas funcionalidades para classes e interfaces, por exemplo, *Add class* e *Add interface*. Estas diagonais têm mais que 75% de similaridade, exceto na linha *Click class button* e na coluna *Click interface button*. A razão para isto é a mesma mostrada na secção 3.2, isto porque ao primeiro clique não havia nada desenhado para ser atualizado. De fato, podemos ver que a semelhança semântica dessas características está atrelada ao alto nível de semelhança de implementação.

3.4. Analisando o relacionamento entre Elementos do Código

A matriz mostrada na Figura 9 foi gerada pela expansão dos nós de elementos do código dos principais pacotes do sistema. Podemos ver um alto número de células pretas. Quando essas células pretas estão relacionadas a elementos do mesmo pacote e mais nenhum outro, esse é um bom indicador para o pacote pois seus elementos estão implementando as mesmas características. Ao mesmo tempo, o fato das características serem implementadas por todos os elementos do pacote, indica um certo grau de espalhamento dessas características pelo pacote. Se nenhum outro pacote implementa essas características isso é um bom sinal, pois a característica encontra-se modularizada no pacote. Entretanto, se mais pacotes implementam essas características isso pode ser um indicador de espalhamento da característica pelo sistema. O pacote `ru` é um exemplo disso. As células vermelhas à esquerda do quadrado azul indicam que os elementos do pacote `ru` não implementam todas as características implementadas pelos pacotes em `org`. Isso ocorre porque as características *Click class button* e *Click interface button* são implementadas em quase todos os pacotes de `org`, mas nunca implementadas no pacote `ru`. Por outro lado, as células pretas acima do quadrado azul indicam que todas as características implementadas por elementos do pacote `ru` são também implementadas pelos elementos de `org` expandidos na figura. Vale destacar a diferença ocorrida na linha para a classe `MFactory`. Selecionando tais células na ferramenta vemos que a classe `MFactory` não implementa as características *Rename class* e *Rename interfaces*, como outros elementos do pacote.

4. Discussão

A escolha das características tem um importante impacto na eficácia da análise. O uso de características similares se mostrou útil na análise de quais são os elementos do código específicos na implementação de cada uma.

Os resultados dos estudos de caso mostrados partiram de rastros de execução específicos. Em uma aplicação com uma interface de usuário complexa, tal como ArgoUML, é virtualmente impossível ter arquivos de rastros idênticos, porque a aplicação reage a muitos eventos de interface e, por exemplo, um movimento de mouse ligeiramente diferente pode desencadear uma seqüência de chamadas também diferente. Entretanto, tivemos o cuidado em realizar várias execuções do mesmo conjunto de características e analisar se as matrizes tinham diferenças significantes. Para os resultados mostrados neste artigo nenhuma diferença significativa pôde ser notada.

De fato, mais experimentações com diferentes sistemas são necessárias. Para sistemas mal projetados, talvez tivéssemos dificuldades em encontrar padrões interessantes nas matrizes e conseqüentemente a compreensão seria mais difícil. Mas neste caso, parece razoável que o processo de compreensão se torne mais difícil de qualquer maneira. O estudo de caso mostrado neste artigo, executa análises de alto-nível, assim não podemos garantir que nossa abordagem é

escalável para análises de granularidade mais fina até o nível de métodos. De fato, a matriz ficará muito grande se os elementos de código forem expandidos completamente. Felizmente, a expansão não tem de seguir todas as direções, sendo possível expandir apenas os métodos de uma classe específica.

A instrumentação normalmente pode influir na execução do sistema. Contudo não notamos nenhuma função anormal que pudesse representar ameaça para os resultados. Como tal observação é específica para este estudo de caso, não podemos garantir isto para outros sistemas, especialmente aqueles com mais *threads* interdependentes.

5. Trabalhos relacionados

As técnicas de localização de características no código fonte têm sido úteis na execução de análises dinâmicas orientadas às características do sistema. O trabalho seminal Software Reconnaissance [11] formou a base de muitos trabalhos subseqüentes nesta área, caracterizada essencialmente pela combinação de técnicas de análise dinâmica e o conceito de características [9, 13, 10,12].

Eisenbarth et al. [9] combina técnicas de análise dinâmica e estática com a análise formal de conceitos para localizar características e explorar as relações entre conjuntos de características e de elementos do código-fonte. Propomos a combinação de diversas visões simultâneas envolvendo a estrutura do código fonte, o modelo de características, e o relacionamento entre pares desses elementos. Greevy e Ducasse [3] tratam o problema de localização de característica sob duas perspectivas complementares: características e unidades computacionais. Para isso, várias métricas são extraídas dos rastros de execução, semelhantes às nossas métricas apresentadas. Por outro lado, o tratamento de múltiplas *threads* e a possibilidade de explorar simultaneamente múltiplos níveis da granularidade de elementos de código é diferente do que fizemos, especialmente no esquema de visualização. A ferramenta que propomos é capaz de alternar rapidamente entre uma visão de alto-nível do sistema e uma visão detalhada. Outra vantagem do nosso trabalho é o uso de abstrações conhecidas das linguagens de programação, tal como, pacotes, classes e métodos para guiar a exploração. Posteriormente, Greevy [10] compila várias técnicas e trabalhos de análise dinâmica focados em características. Nossa idéia de utilizar matrizes coloridas e intervalos de classificação para caracterizar as relações partiu do esquema de coloração usado por Greevy ao definir o grau de afinidade entre elementos de código e características.

O trabalho de Kothari et al. [13] propõe a minimização do número de características consideradas durante a análise de um sistema, visando a redução no esforço de compreensão. Para isso, um conjunto canônico de características deve ser encontrado e validado ao longo de várias versões do sistema. Este representa o conjunto mínimo de características suficiente para a compreensão de todo o sistema. A análise da semelhança entre características é o ponto comum com o nosso.

6. Conclusão

Neste trabalho mostramos como o uso de uma ferramenta de visualização pôde fornecer informações úteis sobre o espalhamento de características e ajudar desenvolvedores a obterem uma melhor compreensão do sistema em tarefas de manutenção, que de outra maneira seria improvável pela simples análise do código e documentação.

A nossa ferramenta ajudou a identificar e entender várias *threads* iniciadas à partir de uma aplicação interativa, ArgoUML. Também pudemos encontrar facilmente informações sobre a intersecção em diferentes níveis de detalhamento entre elementos do código e características. Informações enganosas extraídas da semântica de identificadores levaram a associações aparentemente incoerentes entre características e elementos do código. Nossa ferramenta ofereceu

uma maneira rápida para entender o comportamento real do sistema e constatar que não havia discrepância. Também pudemos identificar facilmente características implementadas com baixo espalhamento, podendo esta informação ser usada em futuras refatorações do sistema.

Contudo 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 [14], esperamos 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 [2], esperamos reduzir a influência de eventos desinteressantes causados por laços, chamadas recursivas e seqüências redundantes. A análise de escalabilidade da abordagem e da ferramenta também deve ser considerada.

***Agradecimentos*:** Agradecemos às agências CAPES e FAPEMIG processo CEX534/03 por financiar parcialmente esta pesquisa.

7. Referências

[1] B. Cornelissen, et a., "Understanding execution traces using massive sequence and circular bundle views," in proc. of Intl.Conf. on Program Comprehension (ICPC'2007), pp. 49–58.

[2] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in proc. of the 14th. Intl. Conf. on Program Comprehension, 2006, pp. 181–199.

[3] O. Greevy and S. Ducasse, "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, pp. 314–323.

[4] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency," in Proc. of the 8th European Conf. on Software Maintenance and Reengineering, 2004.

[5] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman, "Execution patterns in object-oriented visualization," in Proc. of 4th USENIX Conf. on Object-Oriented Tech. and Systems, 1998.

[6] R. Walker, et al. "Visualizing dynamic software system information through highlevel models," in Proc. of OOPSLA, 1998, pp. 271–283.

[7] D. Jerding, J. Stasko, and T. Ball, "Visualizing interactions in program execution," in Proc. of the 19th Intl. Conf. on Software Engineering, 1997, pp. 360–370.

[8] D. Lange and Y. Nakamura, "Object-oriented program tracing and visualization," IEEE Computer, vol. 30, no. 5, pp. 63–70, 1997.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," IEEE Transactions on Software Engineering, vol. 29, no. 3, pp. 210–224, March 2003.

[10] O. Greevy, "Enriching reverse engineering with feature analysis," Ph.D. dissertation, University of Berne, May 2007.

[11] N. Wilde and M. Scully, "Software reconnaissance: mapping program features to code," Software Maintenance: Research and Practice, vol. 7, no. 1, pp. 49–62, 1995.

[12] E. Wong, S. Gokhale, and J. Horgan, "Quantifying the closeness between program components and features," Journal of Systems and Software, vol. 54, no. 2, pp. 87–98, 2000.

[13] J. Kothari, et al. "Reducing program comprehension effort in evolving software by recognizing feature implementation convergence," in Proc. of the ICPC, 2007.

[14] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," ACM SIGPLAN Notices, vol. 40, no. 10, 2005.