Um Experimento na Indústria para Mineração Visual de Padrões de Interações de Programadores

Igor Maciel[1], Methanias Colaço Júnior[1,2], Manoel Mendonça[2], Glauco Carneiro[2]

[1]DSI – Universidade Federal de Sergipe (UFS) – Itabaiana – SE – Brasil

[2]LES – Universidade Federal da Bahia (UFBA) – Salvador – BA – Brasil

igorsmaciel@gmail.com, mjrse@hotmail.com, mgmendonca@dcc.ufba.br, glccarneiro@gmail.com

Resumo. *As ferramentas de visualização de software têm o objetivo de tornar a compreensão de software mais efetiva, embora esta não seja uma atividade trivial tanto para programadores experientes como para os iniciantes. Este artigo utiliza recursos de mineração de dados para analisar e visualizar estratégias adotadas na execução de atividades de compreensão de software apoiadas por ferramentas de visualização. Os resultados revelam as estratégias com os melhores valores de precisão e revocação.*

Abstract. *Software visualization tools have the goal to make software comprehension activities more effective, although it is not a trivial task for both expert and novice programmers. This paper presents a study that uses data mining to analyze and visualize strategies to perform software comprehension activities supported by a visualization tool. The results uncover strategies with better values of precision and recall.*

1. Introdução

A compreensão de um software é a atividade que leva maior tempo durante o processo de manutenção ou evolução do mesmo [Carneiro, Magnavita e Mendonça, 2009]. Este fato está intimamente ligado à necessidade que os programadores têm de conhecer a arquitetura do software e ao mesmo tempo utilizar os seus conhecimentos e experiências passadas para desenvolver suas tarefas de manutenção [Maletic e Kagdi, 2008].

Neste contexto, a utilização de ferramentas visuais para otimizar as tarefas de compreensão tem sido uma prática comum nos processos de manutenção [Maletic, Marcus e Collard, 2002]. Para Moody (2009), notações visuais formam uma parte integral da linguagem da engenharia de software e têm dominado a pesquisa e prática desde os primórdios.

Atualmente, é possível encontrar *plug-ins* de visualização de software para ambientes de programação (IDEs) que fornecem uma representação visual da modularização, hierarquia e dependências de um sistema. Nestes ambientes, utilizando um *plug-in* visual ou não, programadores de diversos níveis utilizam seus conhecimentos e experiência para melhor desenvolver suas atividades. Para realização de uma tarefa, com ou sem sucesso, há uma heurística, isto é, um caminho percorrido que se acredita ser o mais eficiente.

Para alguns *plug-ins*, o uso dos recursos visuais e tradicionais do IDE é totalmente registrado em *logs* durante a tarefa de manutenção [Carneiro, Magnavita e Mendonça, 2009]. Isto permite que as pesquisas em compreensão e visualização de software possam focar em como programadores novatos e experientes realizam algumas tarefas, detectando bons e maus padrões de interação.

Neste artigo, é apresentada uma infraestrutura para mineração visual dos *logs* gerados pela ferramenta de visualização de software *SourceMiner* [Carneiro, Magnavita e Mendonça, 2009]. Estes *logs* foram gerados a partir de um experimento na indústria que explora a identificação de "código mal feito" em softwares, produzindo o insumo suficiente para descoberta de padrões de interação e inferência de possíveis estratégias adotadas pelos programadores.

O restante do artigo é organizado da seguinte forma: a próxima seção discute trabalhos relacionados. A seção 3 descreve o experimento, bem como a infraestrutura criada. Finalmente, na seção 4, são apresentadas conclusões e oportunidades de pesquisas futuras.

2. Trabalhos Relacionados

Alguns pesquisadores têm utilizado mecanismos para registrar heurísticas de programadores. Por exemplo, Bohnet e Döllner (2003) utilizaram recursos de gravação de vídeo para obtenção de dados que descrevem o momento de desenvolvimento do conhecimento e sua aplicação na interface do software. Maletic e Kagdi (2008) também buscaram obter informações sobre o processo cognitivo de compreensão, através da medição por rastreamento ocular.

Em [El-Ramly e Stroulia, 2004], é apresentada a mineração dos dados gerados a partir do uso de um software. Esses dados consistem de uma sequência temporal de eventos que são registrados enquanto o usuário interage com o sistema. El-Ramly e Stroulia (2004) chamam tais sequências de "traços de interação" ou padrões interessantes das atividades do usuário.

Este trabalho seguiu a linha da pesquisa feita por El-Ramly e Stroulia (2004), mas buscou uma inovação, uma vez que a referida pesquisa explorou *logs* de um sistema aplicativo, essencialmente diferente de um ambiente de programação. Além disso, buscamos apresentar o conhecimento visualmente.

3. Experimento

3.1. Definição do Objetivo

O principal objetivo do estudo é criar uma infraestrutura para avaliar e identificar padrões de uso e estratégias aplicadas por programadores na utilização de uma ferramenta de visualização de software acoplada a um IDE. Esse objetivo é formalizado utilizando o modelo GQM proposto por Basili [Basili e Weiss, 1984]: **Analisar** programadores da indústria, **com a finalidade de** avaliação, **em relação aos** padrões de interações, **do ponto de vista de** pesquisadores de engenharia de software, **no contexto de** utilização das ferramentas SOURCEMINER e ECLIPSE.

3.2. Planejamento

O experimento tem como alvo programadores de projetos de código fechado. Os programadores serão convidados a identificar *Code Smells* com a utilização de um ambiente de desenvolvimento enriquecido com recursos de visualização software.

3.2.1. Seleção de Contexto

Nós pretendemos explorar se existe um padrão de sequência para os programadores que obtiverem um maior desempenho. Considerando a dificuldade de se obter a liberação de programadores da indústria para realização de experimentos, a amostra é pequena. Conseqüentemente, devido ao baixo número de programadores liberados pela empresa parceira, um teste estatístico formal não foi executado.

A escolha dos programadores foi por conveniência. Os autores deste artigo conseguiram a liberação de cinco programadores de uma empresa da qual os mesmos são consultores.Por questões legais, não utilizaremos os nomes dos participantes neste trabalho. Letras serão utilizadas para identificar cada desenvolvedor. A Tabela 1 lista estes programadores junto com duas medidas de experiência em manutenção de software.

Tabela 1. Experiência dos programadores disponibilizados

Programador I	Anos de experiência em manutenção	Número de sistemas já mantidos
I	3	6
J	2	3
L	3	5
M	2	2
N	3	6

3.2.2. Instrumentação

3.2.2.1. SourceMiner

O SourceMiner [Carneiro et al., 2009] é um plug-in para o ambiente de desenvolvimento Eclipse que auxilia programadores no processo de visualização e compreensão de software. Os recursos visuais do SourceMiner se combinam com os recursos visuais já fornecidos pelo ambiente Eclipse.

A versão utilizada do SourceMiner integrava os seguintes paradigmas de visualização: (1) *TreeMap*: representa a estrutura hierárquica dos pacotes, classes e métodos de um projeto de software através de retângulos recursivamente aninhados; (2) *Polymetric*: é uma representação bidimensional que usa retângulos arrumados em forma de árvore para representar herança entre entidades do software; (3) *Dependency*: representa dependências entre classes ou pacotes através de grafos radiais; (4) *GridCoupling*: possui dois objetivos: (a) dar uma visão geral sobre o grau de acoplamento dos módulos do software; (b) detalhar o grau de acoplamento de um nó selecionado com outros.

As entidades representadas nas visões do SouceMiner podem ser dinamicamente filtradas de acordo com suas propriedades. Para isto, o plug-in utiliza duas outras visões: FilterView e ConcernFilterView.

3.2.2.3. Logs

O SourceMiner monitora o IDE e provê logs contendo as ações realizadas pelos programadores durante o processo de compreensão e manutenção de um software. Ele

registra cada ação feita pelo usuário no IDE e as armazena em um arquivo texto com os seguintes atributos: data-hora, a *feature* utilizada (recurso usado, o qual pode ser uma visão ou outra janela do IDE) e a ação que foi aplica a esta *feature*. O arquivo forma um histórico estruturado que registra seqüencialmente todas as ações tomadas pelo usuário no ambiente. Este histórico reflete os passos tomados pelo programador durante atividades de engenharia de software suportadas pelo IDE.

3.2.2.4. ETL (Programa de Extração, Transformação e *Load*, ou Carga) para o SourceMiner

Para que os logs gerados pelo SourceMiner pudessem ser efetivamente utilizados, seguindo a mesma arquitetura de Software Data Warehousing apresentada anteriormente em [Colaço Jr. et al., 2009], foi criado um novo ETL em C#. Este programa limpa, transforma e carrega os dados dos *logs* para um *data mart* passível de ser explorado por algoritmos de mineração de dados e por operações OLAP.

O ETL recebe como entrada um arquivo texto com os logs, transformando-os para criação e carga do *data mart* em um banco de dados SQL SERVER. Como o SourceMiner não organiza o *log* em transações, o ETL particiona os dados do log em blocos contendo operações que aconteceram a cada X minutos. Este parâmetro foi definido baseando-se na clássica abordagem da janela deslizante [Fogel e O'Neill, 2009]. Nela, dois eventos subsequentes Ei e Ei+1 fazem parte de uma transação Δ, se os mesmos foram executados em um determinado intervalo de tempo.

Nós fizemos uma média geral do tempo gasto por acesso às *features* do SourceMiner e do Eclipse, multiplicando o mesmo pelo número de *features* existentes no ambiente. O valor final aproximado foi de 4 minutos, o qual foi endossado pelos programadores. Após a entrada do tempo a ser considerado para uma transação, o algoritmo particiona os registros em grupos, os quais contêm as ações que aconteceram dentro do intervalo de tempo que foi estabelecido. Para cada grupo, é associado um número inteiro, único e seqüencial (o número da transação).

Por fim, o *data mart* é carregado, disponibilizando dados tais como o nome da funcionalidade, tempo gasto, nome do programador e o número da transação.

3.2.2.5. Modelo de Mineração de Dados

Uma vez construído o *data mart*, o próximo passo é analisar os seus dados. Além dos resultados de consultas OLAP básicas (dados quantitativos), este experimento também analisará as seqüências de uso das funcionalidades pelos programadores. Para isto, foi utilizado o algoritmo Microsoft Sequence Clustering [MSDN, 2008], disponibilizado pelo pacote de Business Intelligence do SQL Server. Este algoritmo pode ser usado para explorar dados com fatos que podem ser ligados por caminhos ou seqüências.

Esta abordagem localiza as seqüências mais comuns de ações, agrupando (em grupos ou clusters) as seqüências que são similares. Os dados devem ser disponibilizados em forma seqüencial. Em outras palavras, devem representar uma série de eventos ou transições de estados, como os que são gerados pelo ETL desenvolvido para o SourceMiner.

O algoritmo examina todas as probabilidades de transições, medindo a diferença, ou distância, entre as possíveis seqüências no conjunto de dados. De posse dessas diferenças, determina quais seqüências são as melhores ou mais freqüentes.

3.3. Operação

3.3.1. Execução

Para sistematização de uso do SourceMiner por parte dos programadores e conseqüente geração dos logs, foi feita a replicação do experimento apresentado em [Carneiro et al., 2010]. Neste experimento, o objetivo é medir a precisão da identificação de *Code Smells* com a utilização do SourceMiner [Carneiro et al., 2010]. *Code Smells* são anomalias de modularidade de software geralmente causadas pela maneira que interesses são implantados no código fonte [Fowler, 1999].

Os participantes foram solicitados a analisar cinco versões de um sistema de código aberto, para identificar o seguinte conjunto de *Code Smells* [Fowler, 1999]: (1) *Feature Envy* (FE); (2) *God Class* (GC); (3) *Divergent Change* (DC); (4) *Shotgun Surgery* (SS).

Como orientação básica, foi dito aos programadores que os mesmos deveriam tentar descobrir os *Code Smells* sem acessar o código, ou seja, utilizando apenas as visões do SourceMiner e do IDE (sem utilizar o seu Editor). Em último caso, se não fosse possível identificar apenas com as visões, o programador poderia acessar o código da aplicação.

Após a realização desta parte do estudo, foram coletados os logs, os quais foram submetidos ao ETL desenvolvido para o SourceMiner. Carregado o *data mart*, foram executadas consultas OLAP de tempo e número de acessos às *features* do IDE.

Finalmente, cada programador foi individualmente entrevistado e analisado qualitativamente sobre o seu perfil de trabalho na empresa. Os dados da execução foram integrados, validados, analisados e interpretados.

3.3.2 Resultados

A análise dos dados dos *logs* gerados a partir da replicação do experimento de detecção de *Code Smells* se iniciou com consultas OLAP feitas ao *data mart* montado. A Tabela 2 é um exemplo das consultas geradas, as quais apresentam os tempos totais de acesso dos programadores às *features* do SourceMiner e do IDE, bem como as médias por acesso dos programadores a estas *features*.

Essas Tabelas são importantes para evidenciar os programadores que tiveram uma maior dependência de uso do *EDITOR*. A ordem crescente, em minutos, de dependência de uso do Editor foi: L **(1,43)**, I **(3,43)**, M **(14,95)**, J **(29,93)** e N **(30,38)**. Sendo L, o menos dependente do código, e N, o mais dependente.

Tabela 2.Tempos e acessos - Programador I

Programador 1			
Feature	Tempo (minutos)	Acessos	Média (segundos)
GridCoupling	24,22	68	21,37
Dependency	21,98	39	33,82
PACKAGE EXPLORER	10,82	48	13,53
TreeMap	7,87	35	13,49
Polymetric	5,08	32	9,53
ConcernFilterView	4,12	41	6,03
EDITOR	3,43	13	15,83
Filters	2,82	21	8,06
Hierarchy	0,00	0	0,00
Total	80,34		

A Tabela 3 é um exemplo das tabelas geradas que mostram a quantidade de acertos e erros cometidos pelos participantes na identificação dos *code smells* durante o experimento. Com esses valores, são calculadas a precisão e a cobertura [Van Rijsbergen, 1979], bem como a média harmônica das duas medidas, combinando-as em um único valor. A precisão quantifica a porcentagem de respostas dadas que foram corretas, ou seja, quantos *code smells* foram identificados dentre os que realmente existiam. A cobertura quantifica a porcentagem de respostas corretas que foram dadas, sobre todas as respostas corretas, ou seja, a quantidade de acertos dentre os *code smells* existentes.

Considerando o desempenho a partir da média harmônica geral, a ordem decrescente foi: I **(35%)**, L **(30%)**, N **(29%)**, J **(18%)** e M **(14%)**. Sendo I, o melhor desempenho, e M, o pior.

Tabela 3. Precisão, cobertura e média harmônica - Programador I

Programador I	*Code Smells*				Geral
	FE	GC	DC	SS	
Existentes	11	9	15	7	42
Acertos	2	4	5	0	11
Erros	3	1	0	5	9
Precisão	0,4	0,8	1	0	0,55
Cobertura	0,18	0,44	0,33	0	0,26
Média harmônica	0,25	0,57	0,5	0	0,35

Finalizando, a Figura 1 apresenta os resultados de aglomeração de seqüências minerados dos logs. Esta figura representa respectivamente as seqüências de passos mais utilizadas pelos programadores I, J, L, M e N durante as tarefas de compreensão.

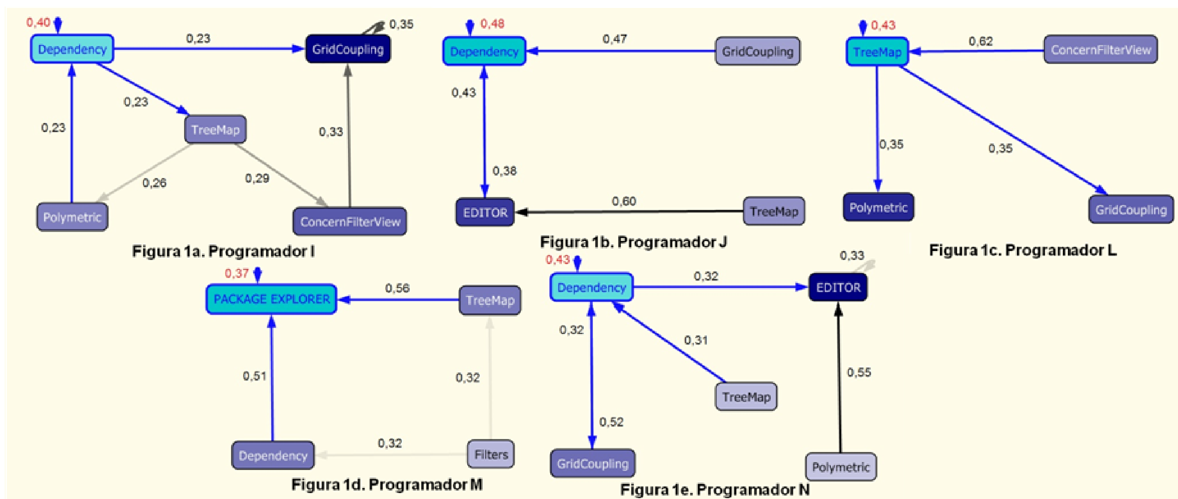

Figura 1. Padrões de Sequência dos programadores

Cada retângulo das figuras representa um estado da seqüência, ou seja, uma *feature* acessada. Os retângulos que possuem sobre eles uma ponta de seta representam as *features* que têm uma probabilidade maior de iniciar uma seqüência. Esta probabilidade pode ser vista no número (cor vermelha) localizado ao lado de cada um deles. As setas que partem de um retângulo para outro representam a transição de estados de uma seqüência; o número ao lado da ponta de cada seta (com a cor preta) representa a porcentagem de vezes que há uma transição entre os estados ligados pela seta. Por exemplo, na Figura 1a, 40% das seqüências são iniciadas pela *feature* Dependency. Partindo do estado Dependency, em 23% das vezes, há uma transição para o estado Gridcoupling, e assim sucessivamente.

3.3.2 Análise e interpretação

Os programadores I e N, que obtiveram o melhor resultado na identificação do *code smell Feature Envy*, com média harmônica igual a 0,25, tiveram grande tendência em iniciar as sequências pela *feature Dependency*, 40% e 43%, respectivamente, e em seguida uma transição para o *GridCoupling*, 23% e 52%, respectivamente. Na identificação do *code smell GodClass*, os programadores I, J e N obtiveram uma média harmônica de 0,57. Suas sequências têm em comum uma grande porcentagem de início pela *feature Dependency*, com 40%, 48% e 43% respectivamente, além de envolver transições entre *Dependecy* e *GridCoupling*.

O melhor desempenho para o *code smell Divergent Changes* foi também do programador I, com média harmônica de 0,50. Nenhum dos participantes obteve pontuação na identificação do quarto e último *code smell, o ShotgunSurgery*.

Considerando todos os erros e acertos de cada participante e utilizando todos os *code smells* existentes no experimento, o programador I obteve o melhor desempenho, com média harmônica geral de 0,35, e o programador M o pior, com 0,14 de média harmônica. O programador I, apesar de ser um dos mais experientes, foi o segundo que menos acessou o código, enquanto que o programador M, o mais inexperiente, ficou em terceiro. Seu fraco desempenho pode ter sofrido influência deste viés. A sequência mais frequente do programador I foi: *Dependency - TreeMap - ConcernViewFilters -*

GridCoupling (Figura 1a). Enquanto a mais frequente do programador M foi: *Filters - Dependency - Package Explorer* (Figura 1d).

Analisando os formulários preenchidos durante o experimento, todos informaram que as visões *Dependency* e *Polymetric* são as mais intuitivas, pois fornecem uma estrutura visual de fácil leitura. Entretanto, entre as duas, somente a *Dependency* foi relatada como útil para as tarefas exigidas. A visão escolhida pelos participantes como a mais útil foi a *GridCoupling*, embora a mesma tenha sido também, em princípio, a mais difícil de compreender. A visão *TreeMap* também foi classificada como de difícil entendimento, pelo menos no início.

4. Conclusões e Trabalhos Futuros

Como foi mostrado neste artigo, é possível, com uma infraestrutura adequada, visualizar parte do conhecimento empírico aplicado pelos programadores durante tarefas de manutenção de um software. Os resultados aqui apresentados servem como base para outros trabalhos, enumeramo-los: (1) Apoio ao programador nas tarefas feitas no IDE, otimizando a interface gráfica; (2) Sugestão de melhores estratégias, baseadas em bons padrões de sequência; (3) Detecção de padrões no uso geral de IDEs, independente da utilização do SourceMiner; e (4) Predição do próximo passo durante a utilização de um IDE ou ferramenta de visualização.

Este trabalho foi parcialmente financiado pelo CNPq e pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (http://www.ines.org.br/).

5. Referências

Basili, V. & Weiss, D. “A Methodology for Collecting Valid Software Engineering Data,” IEEE Transactions on Software Engineering, vol.10(3): 728-738, November 1984.

Bohnet, J.; Döllner, J. Planning an Experiment on User Performance for Exploration of Diagrams Diplayed in 2 1/2 Dimensions. Workshop on SE07, 2003.

Carneiro, G De F., Magnavita, R., Mendonça, M. Proposing a Visual Approach to Support the Characterization of Software Comprehension Activities. In 17th IEEE International Conference on Program Comprehension, USA, Miami University, 2009.

Carneiro, Glauco et al. Identifying Code Smells with Multiple Concern Views. Proceedings of the CBSOFT-SBES, 2010.

Colaço Jr. , Methanias, Mendonça, M. G. & Rodrigues, F. Data Warehousing in an Industrial Software Development Environment. In: 33rd Annual IEEE/NASA Software Engineering Workshop, 2009.

El-Ramly, M.; Stroulia, E. Mining Software Usage Data. Proceedings IEE MSR, 2004.

Fogel, K.; O’Neill, M. ChangeLog Script. www.redbean.com/cvs2cl/, accessed in January, 2009.

Fowler, M. Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

Maletic, J.I.; Kagdi, H. "Expressiveness and Effectiveness of Program Comprehension", In Proceedings of the Frontiers of Software Maintenance, CHI, IEEE, p. 31-37, 2008

Maletic, J.I., Marcus, A., Collard, M.L., "A Task Oriented View of Software Visualization", IEEE Workshop on Visualizing Software for Understanding and Analysis, p. 32-40, 2002

MSDN. Sequence Clustering Algorithm. 2008. Disponivel em: msdn.microsoft.com. Acesso: out. 2010.

Moody, L. D. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions On Software Engineering, v. 35, Novembro 2009.

Van Rijsbergen, C. J. Information Retrieval. 2ª. ed. Londres: Butterworths, 1979.