Anais do WBVS 2012

II Workshop Brasileiro de Visualização de Software

23 de Setembro de 2012

Natal, Rio Grande do Norte, Brasil

Apresentação do WBVS

Esta coletânea reúne os trabalhos selecionados para apresentação no II Workshop Brasileiro de Visualização de Software (WBVS 2012). Este workshop é parte das atividades do Congresso Brasileiro de Software: Teoria e Prática que está atualmente em sua 3ª edição. O objetivo do WBVS é reunir pesquisadores da academia e da indústria para discutir resultados relevantes e desafios enfrentados na área de visualização.

O comitê de programa do WBVS é formado por especialistas em Visualização de Software que fazem parte de grupos de pesquisa reconhecidos da área. Este ano o WBVS recebeu 8 artigos completos. Dessas 8 submissões, 6 artigos foram selecionados para apresentação e publicação nesta coletânea. Cada artigo foi avaliado por 3 (três) membros do comitê de programa. Essas avaliações foram a fonte principal para seleção dos artigos. Elas certamente são importantes contribuições na evolução das pesquisas dos autores.

Agradecemos, especialmente, a Comissão Especial de Engenharia de Software da SBC, a organização do CBSoft, na pessoa de Nélio Cacho, ao coordenador de workshops, Sérgio Soares, aos membros do comitê de organização do WBVS, Daniel Castellani e Marcelo Schots e aos autores dos trabalhos submetidos ao WBVS 2012.

Natal, 23 de setembro de 2012

Cláudia Maria Lima Werner
Leonardo Gresta Paulino Murta
Coordenadores do WBVS 2012

Biografia dos Coordenadores

Cláudia Maria Lima Werner é doutora pela COPPE/UFRJ (1992) e professora associada do Programa Engenharia de Sistemas e Computação da COPPE/UFRJ na área de Engenharia de Software desde 1994, onde lidera o Grupo de Reutilização de Software. É também pesquisadora 1D CNPq e atua na área de Engenharia de Software há mais de 15 anos, nos temas Reutilização de Software, Desenvolvimento Baseado em Componentes, Linhas de Produto de Software e Educação em Engenharia de Software. Possui mais de 200 artigos publicados em conferências e periódicos nacionais e internacionais, além de capítulos de livro. É membro da IEEE, SBC e do comitê de programa de diversas conferências nacionais e internacionais. Mais informações podem ser obtidos em http://lattes.cnpq.br/9719247117370600

Leonardo Gresta Paulino Murta é Doutor (2006) e Mestre (2002) em Engenharia de Sistemas e Computação pela COPPE/UFRJ, e Bacharel (1999) em Informática pelo IM/UFRJ. Atualmente é professor do Instituto de Computação da Universidade Federal Fluminense (UFF), bolsista de Produtividade em Pesquisa nível 2 do CNPq e Jovem Cientista do Nosso Estado da FAPERJ. Seus principais interesses de pesquisa são Gerência de Configuração e Arquitetura de Software. Tem mais de 100 artigos publicados em revistas e conferências. É membro da IEEE, ACM e SBC. Mais informações podem ser obtidas em http://lattes.cnpq.br/1565296529736448.

Coordenadores

Cláudia Werner, COPPE/UFRJ
Leonardo Murta, IC/UFF

Comitê de Organização

Daniel Castellani, STI/UFF
Marcelo Schots, COPPE/UFRJ

Comitê Diretivo

Cláudia Werner, COPPE/UFRJ
Glauco Carneiro, UNIFACS
Leonardo Murta, IC/UFF
Manoel Mendonça, UFBA
Sandra Fabbri, UFSCar

Comitê de Programa

Bianchi Serique Meiguins, UFPA
Claudio Sant'Anna, UFBA
Cleidson de Souza, UFPA
Eduardo Figueiredo, UFMG
Glauco Carneiro, UNIFACS
Gustavo Rossi, Universidad Nacional de La Plata
Heitor Costa, UFLA
Joberto Martins, UNIFACS
José Fernando Júnior, ICMC/USP
Manoel Mendonça, UFBA
Marcelo Pimenta, UFRGS
Marco Aurelio Gerosa, IME/USP
Ricardo Argenton Ramos, Univasf
Rosangela Penteado, UFSCar
Sandra Fabbri, UFSCar
Valter Vieira, UFSCar

Avaliadores Externos

Fábio Petrillo
Gustavo Oliva
Juliana Pereira
Mauricio Aniche
Thieres Nardy Dias

Índice

Propondo uma Arquitetura para Ambientes Interativos baseados em Múltiplas Visões

Arleson Nunes Silva, Glauco de Figueiredo Carneiro

Programa de Pós Graduação em Sistemas e Computação
Universidade Salvador (UNIFACS) – Salvador, BA – Brasil

arleson.nunes@unifacs.br, glauco.carneiro@unifacs.br

Resumo. *Este artigo propõe uma arquitetura para ambientes interativos baseados em múltiplas visões (AIMV). Para esta finalidade são apresentados os objetivos e são discutidos os principais conceitos relacionados aos AIMVs. O artigo também relata o desenvolvimento de duas aplicações a partir do AIMV proposto. A primeira com a finalidade de apoio à compreensão de software. A segunda com a finalidade de apoio à análise do desempenho de redes de computadores. O desenvolvimento destas duas aplicações teve o objetivo de analisar a viabilidade da arquitetura proposta.*

1. Introdução

A visualização é um importante meio de compreensão e é fundamental para apoiar a construção de um modelo mental a respeito de uma determinada situação ou realidade (Spence, 2007). A visualização de informação pode ser mapeada como parte de um processo de compreensão com o objetivo de atingir um conhecimento aprofundado a respeito de um tema a partir de um conjunto de dados (Jacobson, 1999). Para cada tipo ou conjunto de dados podem ser selecionadas uma ou mais metáforas visuais para a sua análise. Estas metáforas podem ser então disponibilizadas em um ambiente interativo baseado em múltiplas visões (AIMV) (Baldonado, Woodruff e Kuchinsky, 2000).

Sistemas baseados em múltiplas visões têm sido propostos e utilizados para a análise de um vasto conjunto de tipos de dados. Na maioria das vezes esta análise requer mais de uma metáfora visual para que seja efetiva (Wang, Woodruff e Kuchinsky, 2000). Isto ocorre em função da dificuldade da representação de determinadas características e propriedades inerentemente complexas em uma única visão (Boukhelifa e Rodgers, 2003; Roberts, 2000; Becks e Seeling, 2004). Além disso, múltiplas visões incentivam a construção de conhecimento mais aprofundado a respeito dos dados analisados e evitam interpretações distorcidas que poderiam emergir de uma única visão (Ainsworth, 1999).

As próximas seções deste trabalho estão estruturadas da seguinte forma. A seção 2 descreve ambientes interativos baseados em múltiplas visões. A seção 3 apresenta a proposta de uma arquitetura para um AIMV. A seção 4 apresenta um relato do desenvolvimento de duas aplicações a partir da arquitetura do AIMV proposto. A seção 5 apresenta as considerações finais.

2. Ambientes Interativos baseados em Múltiplas Visões

Ambientes interativos baseados em múltiplas visões (AIMV) oferecem recursos e mecanismos de visualização para apoiar a análise de dados e também a descoberta e o reconhecimento de informação relevante (Spence, 2007; Shneiderman e Plaisant, 2009).

Estes ambientes utilizam mecanismos de coordenação entre as diversas visões que o compõem. As visões podem ser utilizadas de forma combinada para possibilitar a análise de um conjunto de dados através de diferentes perspectivas. Além disso, um AIMV oferece mecanismos de interação que permitem selecionar e ajustar como os dados serão analisados.

2.1.Objetivos de um AIMV

O objetivo de um AIMV é oferecer um grau de liberdade aos usuários para a execução de tarefas diferentes através de recursos de interação e também do uso combinado das representações visuais disponíveis e coordenadas entre si. Um AIMV deve ser flexível o bastante para permitir: (i) selecionar itens individuais ou subconjuntos de interesses a eles relacionados; (ii) manter o foco em determinados atributos; (iii) utilizar uma visão panorâmica ou detalhada de conjuntos de dados; (iv) navegar pelo conjunto de dados; e, por último, (v) controlar o mapeamento dos atributos reais dos dados nas representações visuais (North e Shneiderman, 2000).

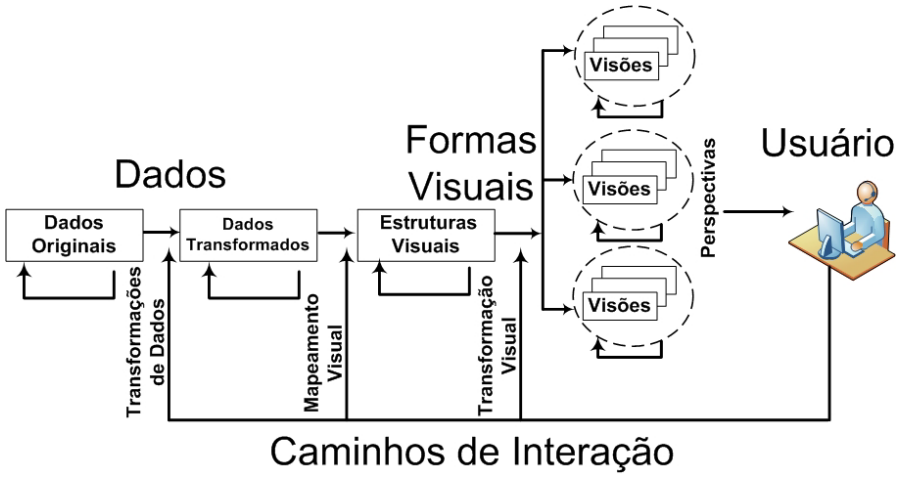


Figura 1: Um Modelo de Referência para AIMVs (Carneiro, 2011)

2.2.Um Modelo de Referência para AIMVs

Um modelo de referência para visualização de informação bastante conhecido foi proposto por Card, Mackinlay e Shneiderman (1999). De acordo com este modelo, a criação de visões ocorre através das seguintes etapas: transformação de dados, mapeamento visual e criação da visão (Card, Mackinlay e Shneiderman, 1999). Carneiro (2011) estendeu este modelo para adaptá-lo ao contexto de múltiplas visões e perspectivas (Figura 1). Uma perspectiva é um conjunto de visões coordenadas que representam um grupo de propriedades ou características específicas de uma situação, contexto ou entidade, geralmente utilizando diversas metáforas visuais (Carneiro, 2011). Este modelo adaptado possibilita o uso combinado dos recursos de um ambiente interativo e baseado em múltiplas visões. O modelo inicia-se com os dados originais obtidos de um repositório. Os dados então passam por um conjunto de transformações para serem organizados em estruturas de dados apropriadas para a exploração. Esta etapa corresponde à transformação dos dados conforme mostrado na Figura 1. Em seguida, estes dados são usados para montar as estruturas de dados visuais. Estas estruturas organizam as propriedades dos dados e propriedades visuais tais como

formas, cores e posições de forma a facilitar a construção das metáforas visuais. Esta etapa corresponde ao mapeamento visual mostrado na Figura 1. A última etapa, a transformação visual, tem o objetivo de representar na tela do computador as informações organizadas na etapa anterior.

2.3.Aplicações Desenvolvidas a partir de AIMVs

Atualmente existem diversas aplicações desenvolvidas a partir de AIMVs. Na área de Engenharia de Software (ES) existe, por exemplo, o Shrimp (Shrimp, 2012), utilizado para representação visual da arquitetura de software. Outro exemplo também em ES é o Tarantula (Tarantula, 2012), utilizado para representação visual de um conjunto de testes para uma determinada aplicação. Existem também exemplos de aplicações baseadas em AIMVs para a análise de dados em outras áreas. Por exemplo, aplicações como InfoZoom (Spenke e Beilken, 2000) e Polaris (Stolte, Tang e Hanrahan, 2002) podem ser utilizadas para a representação visual de dados tabulares com a finalidade de análise de padrões de distribuição.

Entretanto, o que se verifica na prática para a maioria destas aplicações é a dificuldade na inclusão de novos recursos tais como visões ou filtros de interação. Isto poderia auxiliar na configuração dos cenários visuais de forma adequada e compatível às necessidades de análise de um determinado conjunto de dados. Para atender a esta importante demanda, este artigo propõe uma arquitetura de AIMV que viabilize a expansão e customização de aplicações desenvolvidas a partir dela.

3. Propondo uma Arquitetura para um AIMV

Este artigo apresenta uma arquitetura com o objetivo de fornecer uma estrutura básica para o desenvolvimento de aplicações baseadas em múltiplas visões. O principal objetivo desta arquitetura é possibilitar a expansão e customização destas aplicações através da inclusão de novos recursos. Como exemplo, pode-se citar uma aplicação que possui somente uma metáfora visual que representa um conjunto de dados hierárquicos, como, por exemplo, a visão polimétrica (Lanza e Ducasse, 2003). Caso ela tenha sido desenvolvida a partir do AIMV proposto, há a possibilidade de inclusão de outras metáforas visuais para a representação do relacionamento de acoplamento entre os dados, como por exemplo uma visão baseada em grafos (Balzer e Deussen, 2007).

3.1.A Arquitetura Proposta

A arquitetura proposta tem como ponto de partida a arquitetura prosposta por Carneiro (2011). Porém, ajustes foram feitos para que pudesse ser utilizada em diversos domínios de aplicações. A arquitetura tem como referência o modelo de desenvolvimento de Software MVC (Modelo, Visão e Controle), cujo objetivo é separar a lógica de negócio da apresentação ou representação visual (Gamma, *et al.*, 2000). Ela é composta por três camadas (Figura 2). Cada camada tem uma função específica dentro da arquitetura e utiliza um conjunto de componentes independentes para realizá-la.

A primeira camada é a Camada de Modelagem. Essa camada é responsável por importar e modelar os dados em estruturas que facilitem e auxiliem o seu uso pelos recursos da aplicação AIMV. Ela contém os módulos de importação e as estruturas de dados. Os módulos de importação são os componentes responsáveis pela importação e modelagem dos dados nas estruturas de dados, como ilustrado na Figura 2. Eles

recebem o conjunto original dos dados (seta 1) e realizam uma série de transformações para modelá-los (seta 2) de acordo com o objetivo da aplicação. As estruturas de dados são os componentes responsáveis pelo armazenamento dos dados tratados da fonte original. Elas são utilizadas para facilitar e auxiliar o uso dos dados pelos demais componentes da aplicação.

A segunda camada é a Camada de Controle. Ela é responsável pelo controle da aplicação e pela interação entre as estruturas visuais e as estruturas de dados. Ela contém os controladores, os filtros e as ferramentas. Os controladores são componentes responsáveis pelo controle dos eventos gerados pelos demais componentes da aplicação. Eles recebem os eventos (seta 3) e disparam ações de acordo com o tipo de evento recebido e também de acordo com o componente que o enviou. Por exemplo, quando um filtro é aplicado, ele envia um evento para o controlador responsável. Esse controlador vai receber este evento e vai solicitar a atualização (seta 4) das metáforas visuais. Os filtros são os recursos que permitem a filtragem dos dados. Eles atuam diretamente nas estruturas de dados (seta 5) e possibilitam selecionar interativamente um subconjunto de interesse do conjunto de dados. As ferramentas são os recursos que permitem a implementação das funcionalidades da aplicação. Elas também atuam diretamente nas estruturas de dados e podem ser utilizadas para interagir sobre os dados. Tanto os filtros quanto as ferramentas enviam seus eventos para os controladores.

A última camada é a Camada de Visualização. Essa camada fornece os recursos visuais do AIMV. Ela contém os paradigmas (metáforas visuais), as visões de filtragem e a barra de ferramentas. Os paradigmas (parte B da Figura 2) são representações visuais de um conjunto de dados. Eles são resultados da aplicação de uma metáfora visual para representar os dados armazenados nas estruturas de dados em estruturas visuais. As visões de filtragem (parte A Figura 2) são as estruturas responsáveis pela criação dos componentes visuais que serão utilizados para aplicação dos filtros (seta 6). Elas fornecem um conjunto de controles para a execução das operações de filtragem. A barra de ferramentas (parte C da Figura 2) é responsável por fornecer os componentes visuais para aplicar ou ativar as ferramentas (seta 7). Ela armazena os atalhos que dão acesso às ferramentas.

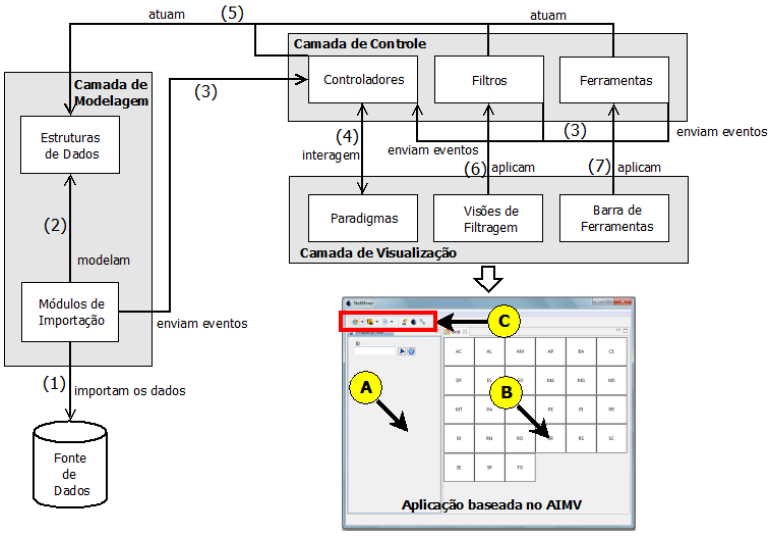


Figura 2: Arquitetura Proposta em Três Camadas para o AIMV

3.2.Estratégias de Desenvolvimento e Evolução

A arquitetura proposta foi desenvolvida como um plug-in do ambiente de desenvolvimento de software (ADS) Eclipse (Eclipse, 2012) usando a linguagem Java. Na concepção da arquitetura foram utilizados recursos de extensão e pontos de extensão do Eclipse (Eclipse, 2012). O objetivo é permitir que os componentes da aplicação possam ser instanciados em plug-ins diferentes. Com isso, aplicações desenvolvidas com base na arquitetura podem ser adaptadas para outros contextos utilizando apenas um conjunto de plug-ins já desenvolvidos. Além disso, os componentes podem ser reutilizados para diferentes objetivos. A Figura 3 representa a arquitetura do plug-in AIMV. Ele contém um conjunto de pacotes, classes e recursos internos que são responsáveis por fornecer as funcionalidades da arquitetura. Além disso, este plug-in oferece cinco pontos de extensão, como mostra a Figura 3. Estes pontos de extensão possuem um conjunto de regras e propriedades para a criação dos componentes da aplicação e facilitam a inclusão de novas funcionalidades.

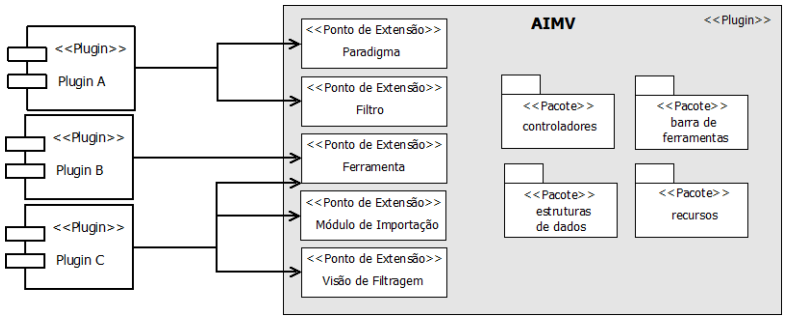


Figura 3: Estrutura do plug-in AIMV

4. Relatando o Desenvolvimento de Duas Aplicações a partir do AIMV Proposto

O objetivo desta seção é analisar a viabilidade do desenvolvimento de aplicações a partir da arquitetura proposta e utilizando a estrutura básica disponibilizada pelo plug-in AIMV mostrado na Figura 3. Para esta finalidade, será apresentado um relato do desenvolvimento pelo primeiro autor deste artigo de duas aplicações: o SourceMiner e o NetMiner conforme descrito a seguir.

4.1.SourceMiner

O SourceMiner[1] é um aplicação do tipo AIMV para representar através de metáforas visuais informações consideradas relevantes de uma aplicação Java (Carneiro, 2011). A aplicação foi desenvolvida como um plug-in do Eclipse com a finalidade de apoio a atividades de compreensão de software. As primeiras versões do SourceMiner tinham alguns de seus componentes com forte acoplamento, fato que tornava mais onerosa a inclusão de novas funcionalidades tais como novas metáforas visuais. Em alguns casos, isto implicou na necessidade de diversas modificações em cascata para que fosse possível a alteração da implementação de determinados componentes. Além disso, para o desenvolvimento de novas funcionalidades era necessário o acesso e uso do código fonte do SourceMiner.

[1] O SourceMiner está disponível em http://www.sourceminer.org.

A nova versão do SourceMiner desenvolvida a partir da arquitetura proposta do AIMV contornou estas questões. Além da implementação das funcionalidades já existentes, também foram tratadas oportunidades de melhorias identificadas em uma avaliação de usabilidade realizada por 15 usuários do SourceMiner. A Figura 4 mostra o relacionamento entre os plug-ins que compõem a nova versão do SourceMiner desenvolvido a partir da arquitetura proposta do AIMV.

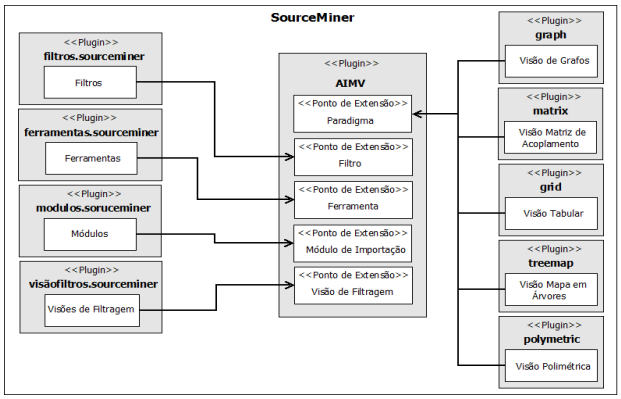


Figura 4: Nova Arquitetura do SourceMiner

O plug-in central *AIMV* da Figura 4 representa o mesmo plug-in indicado na Figura 3. Ele fornece a estrutura básica do AIMV e os pontos de extensão para instanciar os componentes da aplicação, nesse caso o SourceMiner. A partir daí, foram desenvolvidos outros plug-ins para a criação dos componentes. Cada plug-in foi responsável pela criação de um conjunto de componentes específicos do SourceMiner. Neste caso, um plug-in para criação dos filtros, outro para criação das ferramentas, outro dos módulos e por último um plug-in para as visões de filtragem, como mostra a parte esquerda da Figura 4.

Os demais plug-ins, apresentados à direita da figura, foram desenvolvidos para criação das visões. Foram implementadas as mesmas visões presentes na versão anterior do SourceMiner, porém com base na nova arquitetura. A visão de grafos (Balzer e Deussen, 2007), a visão polimétrica (Lanza e Ducasse, 2003), a visão mapa em árvores (Shneiderman, 1992), a visão matriz de acoplamento (Sangal, *et al.*, 2005) e a visão tabular (Carneiro, 2011), representadas respectivamente pelos plug-ins: *Graph, Polymetric, TreeMap, Matrix e Grid*. Como pode ser constatado, o SourceMiner é a aplicação resultante da integração de todos estes plug-ins ilustrados na Figura 4. Além disso, o plug-in AIMV fornece a estrutura básica da aplicação enquanto que os demais implementam as funcionalidades e instanciam os componentes do SourceMiner.

4.2.NetMiner

O NetMiner é um ambiente integrado baseado em múltiplas visões (AIMV) utilizado para apresentar visualmente as métricas de desempenho de redes de computadores. Este ambiente pode permitir a monitoração e a análise de forma visual dos principais atributos que afetam o desempenho das redes. Ele foi desenvolvido a partir da arquitetura proposta, utilizando o plug-in *AIMV*. A Figura 5 mostra a estrutura do NetMiner.

O plug-in central *AIMV* da Figura 5 representa o mesmo plug-in da arquitetura proposta mostrado na Figura 3. Ele fornece a estrutura básica do AIMV e os pontos de

extensão para instanciar os componentes da aplicação, nesse caso o NetMiner. De forma similar ao SourceMiner, foram desenvolvidos outros plug-ins para a criação dos componentes. Cada plug-in foi responsável pela criação de um conjunto de componentes específicos do NetMiner. Além disso, o NetMiner utiliza as mesmas visões utilizadas no SourceMiner. A diferença é que no NetMiner estas visões são utilizadas com um objetivo diferente. O foco agora é no uso de um ambiente interativo baseado em múltiplas visões para a análise das métricas de desempenho de redes de computadores. O NetMiner consiste então na integração de todos esses plug-ins mostrados de acordo com o que é apresentado na Figura 5.

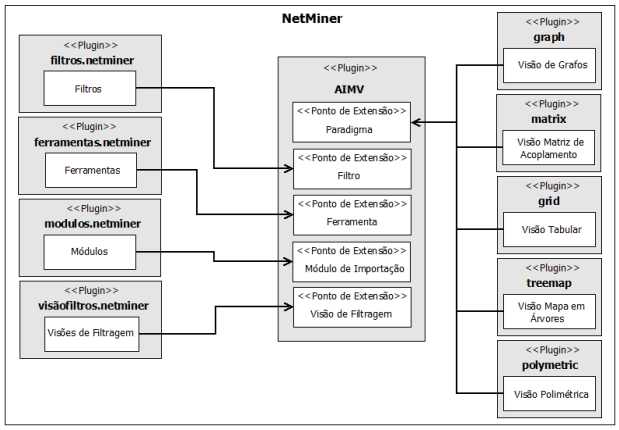


Figura 5: Arquitetura do NetMiner

5. Considerações Finais

Sistemas baseados em múltiplas visões podem ser utilizados para a análise de um vasto conjunto de tipos de dados. Atualmente existem diversas aplicações desenvolvidas a partir de AIMVs em diversas áreas de conhecimento. Entretanto, o que se verifica na prática para a maioria destas aplicações é a dificuldade na inclusão de novos recursos. Neste contexto, o artigo apresentou uma arquitetura que tem como objetivo fornecer uma estrutura básica para o desenvolvimento de aplicações baseadas em múltiplas visões. Esta arquitetura visa possibilitar a expansão e customização destas aplicações através da inclusão de novos recursos. O estudo de caso relatado ilustra o desenvolvimento de duas aplicações para a análise de dois tipos de conjuntos de dados diferentes a partir da arquitetura proposta de AIMV. Os resultados preliminares indicam evidências iniciais que a arquitetura é efetiva para o desenvolvimento de aplicações interativas baseadas em múltiplas visões. Neste momento os autores estão planejando dois estudos. O primeiro com foco na avaliação de usabilidade das versões atuais do SourceMiner e NetMiner. O segundo estudo terá como foco a efetividade do uso destas aplicações para atividades nas áreas de engenharia de software e redes de computadores.

Referências

Spence, R. Information Visualization: Design for Interaction (2nd Edition). 2. ed.Prentice Hall, 2007.

Jacobson, I.; Booch, G.; Rumbaugh, J. The Unified Software Development Process. Addison-Wesley Professional, 1999.

Baldonado, M., Woodruff, A., Kuhinsky, A. Guidelines for Using Multiple Views in Information Visualization, In ACM AVI 2000; Palermo, Italy. 110-119.

Wang, M. Q.; Woodruff, A.; Kuchinsky, A. Guidelines for using multiple views in information visualization. Proceedings of the working conference on Advanced visual interfaces. ACM. 2000. p. 110-119.

Boukhelifa, N.; Rodgers, P. J. A model and software system for coordinated and multiple views in exploratory visualization. Information Visualization, v. 2, p. 258-269, 2003.

Roberts, J. C. Multiple-View and Multiform Visualization. IS&T and SPIE Visual data exploration and analysis. 2000. Vol. 3960 p. 176-185.

Becks, A.; Seeling, C. SWAPit: a multiple views paradigm for exploring associations of texts and structured data. Proceedings of the Working Conference on Advanced Visual Interfaces. ACM. 2004. p. 193-196.

Ainsworth, S. The functions of multiple representations. Comput. Educ., v. 33, p. 131-152, 1999.

Shneiderman, B.; Plaisant, C. Designing the User Interface: Strategies for Effective Human-Computer Interaction (5th Edition), 5th ed. Addison Wesley, 2009.

North, C.; Shneiderman, B. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. Proceedings of the Working Conference on Advanced Visual Interfaces ACM. 2000. p. 128-135.

Card, S. K.; Mackinlay, J. D.; Shneiderman, B. (Eds.). Readings in information visualization: using vision to think. Morgan Kaufmann Publishers Inc., 1999.

Carneiro, G. F. Sourceminer: um ambiente integrado para Visualização multi-perspectiva de software. 2011. 230 f. Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática, Doutorado Multiinstitucional em Ciência da Computação.

Tarantula software. http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/index.html, Maio. 2012.

Shrimp software. http://www.thechiselgroup.org/shrimp), Maio. 2012.

Spenke, M.; Beilken, C. InfoZoom: Analysing Formula One racing results with an interactive data mining and visualisation tool. In Proceedings of 2nd International Conference on Data Mining, pages 455–464, Cambridge University, UK, 2000.

Stolte, C.; Tang, D.; Hanrahan, P. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. IEEE Transactions on Visualization and Computer Graphics, pages 52–65, 2002.

Eclipse. http://www.eclipse.org/, Maio. 2012.

Balzer, M.; Deussen, O. Hierarchy Based 3D Visualization of Large Software Structures. Proceedings of the Conference on Visualization. 2004. p. 4-14.

Lanza, M.; Ducasse, S. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. IEEE Trans. Softw. Eng., v. 29, p. 782-795, 2003.

Shneiderman, B. Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph., v. 11, p. 92-99, 1992.

Sangal, N. et al. Using Dependency Models to Manage Complex Software Architecture. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications. 2005. p. 167-176.

Gamma, Erich et al. *Padrões de Projeto:* Soluções reutilizáveis de software Orientado a Objetos. Porto Alegre: Bookman, 2000.

TimeLine Matrix: an on demand view for software evolution analysis

Renato L. Novais[1,2], Paulo R. M. Simões Júnior[1], Manoel Mendonça[1]

[1]Computer Science Department, Federal University of Bahia (UFBA), Bahia, Brazil

[2] Information Technology Department, Federal Institute of Bahia, Campus Santo Amaro, Bahia, Brazil

{renatoln, pauloroberto, mgmendonca}@dcc.ufba.br

Abstract. *Software evolution is one of the most important topics of modern research in software engineering. Many works of software visualization have been proposed to assist in data analysis of software evolution. These studies usually overview the history or show a snapshot of the evolution. Many of them provide details on demand, but only through tooltips or accessing the source code. We present the vision Timeline Matrix, which operates on demand from other views. Once selected the element of interest, this view shows the complete evolution of this element. Allowing to easily navigating in the history, and making comparisons among up to three elements. An example usage of this view in a real scenario is presented, showing how it can help solve evolution comprehension tasks.*

Resumo. *Evolução de software é um dos tópicos mais importantes da pesquisa moderna em engenharia de software. Muitos trabalhos de visualização de software têm sido propostos para ajudar na análise dos dados de evolução. Estes trabalhos, normalmente, apresentam uma visão global ou um retrato instantâneo da evolução do software. Muitos deles oferecem detalhes sob demanda, porém apenas através de tooltips ou acesso ao código fonte. Neste trabalho, apresentamos a visão TimeLine Matrix, a qual funciona sob demanda de outras visões. Uma vez selecionado o elemento de interesse, esta visão mostra a evolução completa deste elemento. Ela permite assim navegar facilmente na história, e fazer comparações entre até três elementos. Um exemplo de uso dessa visão em um cenário real é apresentado, mostrando como ela pode ajudar a resolver tarefas de compreensão de evolução de software.*

1. Introdução

Evolução de software é um dos tópicos mais importantes da pesquisa moderna em engenharia de software. Esta atividade requer a análise de grande quantidade de dados que descrevem sua estrutura atual, bem como sua história. As áreas de métricas e visualização de software têm sido utilizadas para facilitar a compreensão deste cenário. As métricas ajudam a medir e controlar os artefatos que estão evoluindo [Wohlin et al. 2000]. A visualização ajuda a sumarizar os dados complexos em compreensíveis cenários visuais [Diehl 2007][Storey 2005].

As duas áreas são comumente utilizadas em conjunto. Os números apresentados apenas através dos seus valores não são intuitivos para os usuários. A visualização faz uso de metáforas visuais que facilitam a interpretação dos valores das métricas.

Muitos trabalhos de visualização de software têm sido propostos para ajudar na análise de dados de evolução [Eick et al. 1992] [Lanza 2001] [D'Ambros et al. 2009] [Cepeda et al. 2010][Bergel et al. 2011]. A grande maioria mostra os dados através de: i) visões globais (*overview*), que apresentam, na mesma cena visual, toda a história do software, e, ii) visões instantâneas (*snapshots*), que mostram o software em algum momento da história. Estes trabalhos seguem o mantra de visualização de Shneiderman (1996): *Overview first, details on demand*. Entretanto, a parte de "detalhes sob demanda" destes trabalhos são, normalmente, interagir com as visões para acessar mais propriedades do módulo de software na versão de interesse, ou até mesmo acessar o código fonte dessa versão.

Nos últimos três anos, temos trabalhando num ambiente de visualização de evolução de software chamado *SourceMiner Evolution* – SME[1] – [Novais et al. 2011a][Novais et al. 2011b][Novais et al. 2012]. Esse ambiente permite visualizar a evolução de software seguindo diferentes Estratégias de Análise[2] de evolução de software através recursos visuais. Até esse momento, estão disponíveis a estratégia diferencial relativa [Novais et al. 2011a], estratégia diferencial absoluta [Novais et al. 2012], estratégia temporal overview [Novais et al. 2011b]. Apesar da gama de recursos disponíveis (e.g. detalhes sob demanda, da forma como citado anteriormente), percebemos, nos diversos estudos realizados, a necessidade de uma visão que fornecesse detalhes sob demanda de uma forma diferente das atualmente propostas na literatura. A ideia é poder selecionar um elemento de software de interesse nas outras visões, e poder visualizar a sua evolução separadamente, vendo todo o histórico do elemento de interesse.

Nesse contexto, desenvolvemos uma nova visão: *TimeLine Matrix* (TLM). Esta visão é altamente configurável, permitindo visualizar a evolução de até três elementos de software por vez. Os atributos visuais podem ser facilmente mapeados aos atributos reais disponíveis no ambiente (e.g. métricas de software, mapeamento de funcionalidades (*features*) no código fonte). Desta forma, é possível fazer a análise da evolução do software por elemento, podendo inclusive fazer a comparação da evolução de até três elementos de software diferentes.

Além dessa introdução, este artigo está organizado como se segue. Na Seção 2, serão apresentados alguns trabalhos relacionados. Na Seção 3, será apresentada essa nova visualização desenvolvida, mostrando todos os recursos disponíveis. Exemplo de uso, baseado em um dos nossos estudos anteriores, é apresentado na Seção 4. Por fim, a Seção 5 conclui este artigo.

---

[1] Website do SME: http://www.sourceminer.org/

[2] Uma estratégia de análise define como a evolução de um artefato de software é apresentada para análise. A representação dessa evolução pode ser representada por diferentes maneiras que facilitam (ou dificultam) a compreensão dos fenômenos associados às mudanças. Não é objetivo desse trabalho explicar estas estratégias. Para um melhor entendimento sugerimos nosso trabalhos anteriores, cujas referências estão disponíveis no texto.

2. Trabalhos Relacionados

A visualização tem sido bastante utilizada para análise de evolução de software. Ao longo do anos, muitos trabalhos têm sido propostos. Trabalhos com diferentes visões, estratégias de análise, perspectivas, e focando em tarefas diferentes de Engenharia de Software.

Visualização de evolução de software não é algo recente. Um dos primeiros trabalhos é datado de 1992, por Eick et al. O *SeeSoft*, proposto por estes autores, mapeia cada linha de código a linhas formadas por pixels. Dentre as suas funcionalidades, estão desde a compreensão de uma versão, quanto a visualização de mudanças recentes no código fonte. Outro trabalho que tem uma importância histórica bastante relevante, é o *Evolution Matrix* [Lanza 2001]. Neste trabalho, o autor propõe uma matriz de evolução mostrando todas as classes do sistemas em todas as versões disponíveis.

Mais recentemente, muitos trabalhos foram publicados com diferentes propósitos. D'Ambros et al. (2009) propuseram uma visualização em radar para analisar acoplamento lógico entre os módulos do software. Cepeda et al. (2010) propuseram uma visualização que permite detectar e externalizar a evolução do design do software. Em [Bergel et al. 2011] é apresentada outra visualização que permite comparar perfis de diferentes versões do software e destacar mudanças críticas de desempenho no sistema.

A *TimeLine Matrix* segue a mesma linha da *Polymetric Views* de Lanza e Ducasse (2003), permitindo o mapeamento de atributos reais a atributos visuais. Do ponto de vista de paradigma visual ela se assemelha ao *Evolution Matrix*. Porém se difere dela e dos outros aqui apresentados, por ser uma visão sob demanda, integrada a outras visões, e facilmente configurável. Assim, o usuário pode configurar a visualização de acordo com seu objetivo, e da forma que mais facilite a sua compreensão da evolução do software.

3. TimeLine Matrix

A *TimeLine Matrix* (TLM) é uma nova visão que permite a visualização da evolução de elementos de software sob demanda. A Figura 1 apresenta uma visão geral do SME dando ênfase a visão TLM. O SME possui três visões gerais (*Treemap*, *Polymetric*, *Coupling*) que utilizam as estratégias de análise Diferencial Relativa e Diferencial Absoluta. A quarta visão é a Coordenadas paralelas. Ela endereça a estratégia Temporal overview, e também funciona sob demanda a partir das três primeiras visões. A partir de qualquer uma das três visões gerais é possível selecionar um elemento de software de interesse, em qualquer versão disponível e solicitar detalhes sob demanda na visão TLM.

A visão TLM foi projetada no formato de uma matriz 3x3. Cada linha desta matriz é chamada de *timeline* de evolução. Cada coluna representa uma versão do elemento de software em análise. Cada *timeline* mostra por vez até três versões do elemento. É possível selecionar para cada *timeline* um elemento de software diferente ou o mesmo elemento de software para todos os *timelines*. Neste último caso, será possível ver até nove versões do mesmo elemento na mesma cena visual. Recursos de navegação entre as versões estão disponíveis na visão.

Um dos recursos mais importante dessa visão, o qual dá a ela uma característica bastante flexível, é a possibilidade de mapear facilmente atributos reais (e.g. métricas de software) a três atributos visuais disponíveis nessa visão (tonalidade da cor, largura e altura do retângulo). Desta forma, o usuário pode selecionar quaisquer métricas, dentre as disponíveis, e visualizar a evolução de acordo com seu interesse. O crescimento do atributo real é mapeado na coloração mais escura do elemento visual, bem como na maior largura ou altura do elemento visual. O decrescimento é tratado de forma oposta. Três cores distintas são utilizadas nessa visão: azul, vermelho, e verde para representar pacotes, classes e métodos, respectivamente.

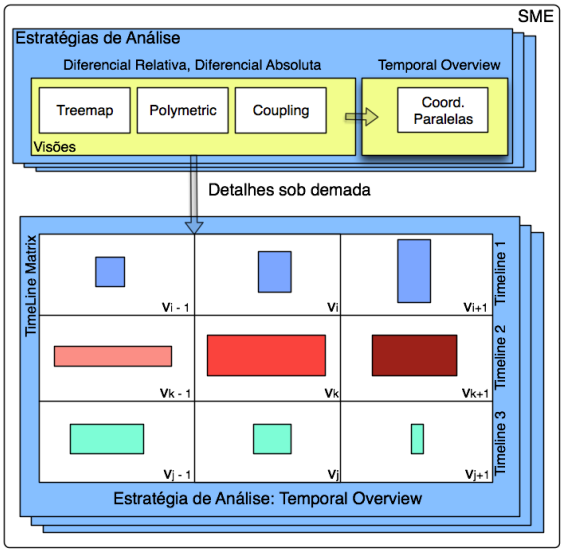


Figura 1. Uma visão geral do SME com ênfase na *TimeLine Matrix*

No *Timeline* 1 da Figura 1, por exemplo, é possível observar que houve um crescimento na métrica mapeada ao atributo visual "altura do retângulo" da versão v_{i-1} até v_{i+1}. No *Timeline* 2 é possível observar que a "tonalidade" do vermelho está escurecendo da versão v_{k-1} até v_{k+1}, de onde conclui-se que a métrica mapeada também está crescendo. Por fim, no *Timeline* 3 há um decrescimento da métrica associada a "largura do retângulo" da versão v_{j-1} até v_{j+1}.

Através dessa nova visão, o usuário pode comparar a evolução de até três elementos diferentes de software. Isso permite análises comparativas e percepção de tendências entre os elementos.

Diversas métricas estão disponíveis para uso na TLM. Como exemplos, podem ser citadas Linha de Código, Número de Métodos, complexidade ciclomática, acoplamento aferente e eferente, Número de classes, etc. Cada atributo de software possui seu conjunto de métricas específico, de tal forma que não é possível mapear atributos reais a atributos visuais de forma inconsistente (e.g. mapear na "largura do retângulo" o número de métodos, quando o elemento sendo visualizado é o próprio método).

A Figura 2 apresenta a visão TLM em ação. É possível observar as nove células da matriz, sendo que sob cada célula, é apresentado a versão do elemento. Botões de navegação *forward* e *backward* existem em cada *timeline*. Por fim, sobre cada *timeline* é

apresentado o nome do elemento sendo visualizado. No lado esquerdo da Figura 2 é apresentado a visão *TimeLine Filter*, que permite fazer os mapeamentos dos atributos reais aos atributos visuais para cada um dos três *timelines*.

Essa visão possui também a possibilidade de se obter mais detalhes sob demanda. Através do *tooltip*, o usuário pode acessar informações detalhadas do elemento sendo visualizado, como por exemplo as funcionalidades (*features*) sendo realizada pelo elemento de software. Visualização de evolução de *features* é um recurso disponível no SME [Novais et al. 2012]. É possível ainda acessar o código fonte a partir dos elementos visuais. Outra funcionalidade é a possibilidade de se fazer o caminho inverso entre as visões. A partir de um elemento (pacote, classe ou método) na TLM, pode ser solicitado, através de menu *pop up*, para que o elemento seja visualizado na visão *Treemap*. Esta operação abre, por exemplo, uma classe em detalhe na *Treemap*, mostrando todos os seus métodos.

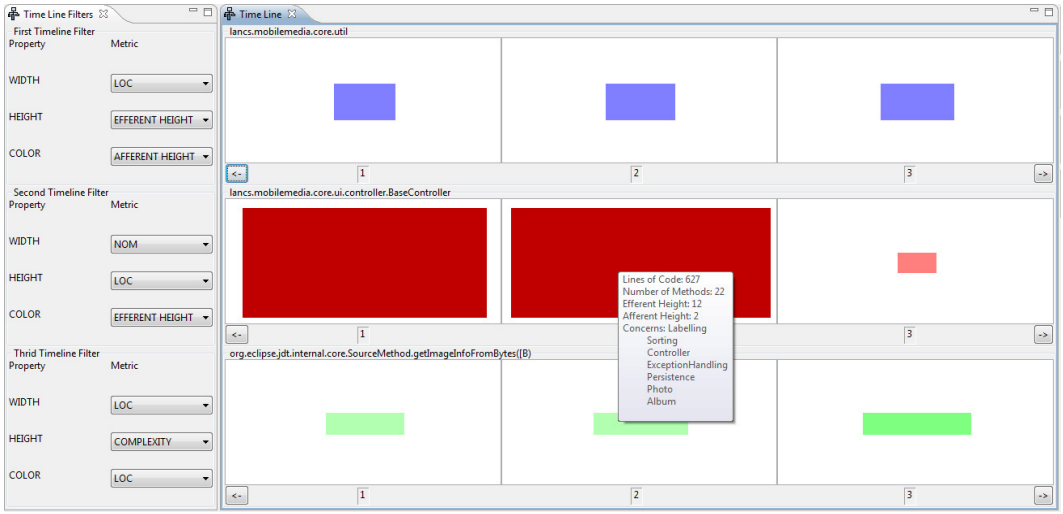


Figura 2 – Visão *TimeLime Matrix* em ação

4. Exemplo de Uso

Nesta seção será apresentado um exemplo de uso da TLM. Esta nova visão surgiu a partir dos estudos realizados anteriormente, principalmente a partir do experimento controlado descrito em [Novais et al. 2012]. Nesse experimento, foi feita uma comparação entre o SME e *ConcernMapper* [Robillard e Weigand-Warr 2005] em relação a análise de evolução de *features*. O estudo contou com a participação de 20 desenvolvedores, sendo 10 de Salvador – Bahia e 10 do Rio de Janeiro – RJ. Os participantes deveriam realizar um conjunto de seis atividades, relacionadas à compreensão de evolução de *features*, em cinco versões de um software industrial. As tarefas estavam classificadas em três objetivos diferentes: Análise de Evolução de *Feature*, Análise de Entrelaçamento de *Feature*, e Análise de Dependência de *Features*.

Duas hipóteses foram definidas para o estudo: H1) O SME decresce o tempo gasto nas tarefas de compreensão de evolução de *features*; H2) O SME melhora a corretude das tarefas de compreensão de evolução de *features*. Após a análise estatística dos dados, pôde-se observar que o SME decresceu o tempo gasto, em média, em 8.95%

em relação ao *ConcernMapper*, entretanto não foi possível rejeitar a hipótese nula, consequentemente não foi possível aceitar a hipótese H1. Por outro lado, em relação a hipótese H2, o SME aumentou a corretude, em média, em 26.94% quando comparado com o *ConcernMapper*. Os resultados, neste caso, foram relevantes, podendo inclusive aceitar a hipótese (H2) de que o SME melhora a corretude.

Diversos fatores relacionado aos tratamentos, ao objeto de estudo, e *design* do experimento foram discutidos a fim de entender os resultados encontrados. Foi observado, por exemplo, que o SME não obteve melhores resultados em relação a eficiência, devido ao tempo gasto para a representação gráfica do software sob estudo, nas visões utilizadas. Como o software era grande, as visões levavam tempo para serem redesenhadas a cada mudança de versão. Isto ficou evidente principalmente em relação a duas tarefas específicas. Essas tarefas são discutidas a seguir, mostrando como os participantes utilizaram o SME durante o experimento, e como seria, caso a visão *TimeLine Matrix* estivesse disponível durante esse experimento. As duas tarefas são relacionadas à análise de entrelaçamento de *features*.

Tarefa T4: Quais são as features realizadas pela classe DBConnection em cada versão?

Para realizar essa tarefa durante o experimento, os participantes tiveram basicamente que realizar um conjunto de ações sobre o SME: i) selecionar a versão 1 para ser visualizada; ii) utilizar o filtro para encontrar a classe *DBConnection* na visão; iii) requisitar detalhes sob demanda (e.g. *tooltip*) para descobrir quais as *features* realizadas por esse elemento de software. Todos esses passos teriam que ser repetidos para cada uma das outras quatro versões seguintes. O grande problema enfrentado foi no momento de redesenhar as visões quando havia a mudança de versão. Isso aconteceu principalmente pelo fato do objeto de estudo ser um sistema de larga escala, com, em média, 500 classes em cada versão.

Em atividades desse tipo, é possível observar que, uma vez encontrado o elemento de software de interesse, não se faz mais necessário visualizar os outros elementos de software. Isto foi melhorado com a implementação da TLM. Utilizando a TLM, o participante teria que realizar os seguintes passos: i) selecionar a versão 1 para ser visualizada; ii) utilizar o filtro para encontrar a classe *DBConnection* na visão; iii) requisitar para que o elemento de interesse fosse apresentado em algum *timeline* da TLM; iv) requisitar detalhes sob demanda na visão TLM, recuperando as *features* para cada versão da classe em questão. O usuário poderia visualizar as cinco versões de uma só vez, utilizando para isso dois *timelines* distintos, ou navegar em um único *timeline* através dos componentes de navegação disponíveis. Neste caso, não existe mais o tempo gasto para o processamento e representação de todos os elementos de cada uma das versões do software sob estudo.

Tarefa T5: A classe ServerConfig está realizando mais de uma feature ao longo da evolução? Se sim, quais são as features? Para cada feature, em cada versão, quais são os métodos realizando ela?

O processo de realização dessa tarefa é bem semelhante ao da tarefa anterior. Para realizá-la durante o experimento, os participantes tiveram basicamente que realizar o seguinte conjunto de ações sobre o SME: i) selecionar a versão 1 para ser visualizada; ii) utilizar o filtro para encontrar a classe *ServerConfig* na visão; iii) requisitar detalhes

sob demanda (e.g. *tooltip*) para descobrir quais as *features* realizadas por esse elemento de software; iv) identificar, para cada método dessa classe, quais *features* eles realizam. Da mesma forma, todos esses passos teriam que ser repetidos para cada uma das outras quatro versões seguintes. O problema de redesenhar as visões quando havia a mudança de versão também impactou no resultado final do experimento.

Essa atividade também seria mais facilmente realizada com a utilização da TLM. Com esta visão, o participante teria que realizar basicamente os seguintes passos da tarefa anterior: i) selecionar a versão 1 para ser visualizada; ii) utilizar o filtro para encontrar a classe *ServerConfig* na visão; iii) requisitar para que o elemento de interesse fosse apresentado em algum *timeline* da TLM; iv) requisitar detalhes sob demanda na visão TLM, recuperando as *features* para cada versão da classe em questão. A principal diferença aqui, é que neste caso é necessário identificar as *features* por método, e o TLM estaria apresentando a classe. Para resolver esse problema, o SME dispõe de um recurso de integração entre todas as visões. Neste caso, deve-se solicitar para que a classe de interesse seja visualizada na visão *Treemap*. Assim, as *features* por métodos seriam coletadas na visão *Treemap*, enquanto que a navegação entre as versões seria realizada através do TLM. Desta forma, o uso do TML permitiria novamente a diminuição do tempo gasto com o redesenhar das visões.

5. Conclusão

Dada a importância da evolução de software, diversos trabalhos de visualização de evolução de software têm sido propostos. Estes trabalhos oferecem ricas visões, com possibilidade de acesso a detalhes sob demanda, através de *tooltips* ou acesso ao código fonte. Entretanto, algumas tarefas de compreensão de evolução de software requerem um outro tipo de detalhes sob demanda: uma vez selecionado um elemento de interesse numa visão global, poder visualizar a evolução apenas desse elemento.

No contexto desse trabalho, foi apresentado a visão *TimeLine Matrix*. Uma visão sob demanda que está integrada ao ambiente de visualização de evolução de software *SourceMiner Evolution*. Esta nova visão permite que o usuário navegue na história de um elemento de software (pacote, classe ou método) de forma rápida e prática. Mapeamentos de recursos reais a recursos visuais são disponíveis na visão, tornando-a bastante flexível. Desta forma, o usuário pode configurá-la de acordo com o sua tarefa de compreensão de evolução.

Para mostrar sua aplicabilidade, foi apresentado como a *TimeLine Matrix* ajudaria a resolver duas tarefas de compreensão de evolução de *features*, extraídas de um estudo controlado realizado anteriormente. Entretanto, é sabido que sua efetividade ainda precisa ser avaliada. Isto está sendo planejado, por exemplo, através da replicação do estudo apresentado. Espera-se que os usuários do SME se beneficiem com as funcionalidades disponibilizadas através da nova visão *TimeLine Matrix*.

Referências

Bergel, A., Bañados, F., Robbes, R. and Binder, W. (2011), “Execution profiling blueprints” In Softw: Pract. Exper.

Cepeda, R. D. S. V., Magdaleno, A. M., Murta, L. G. P., Werner, C. M. L. (2010). “EvolTrack: Improving Design Evolution Awareness in Software Development”, In

Journal of the Brazilian Computer Society (JBCS), Volume 16, Number 2 (2010), 117-131.

D'Ambros, M., Lanza, M. and Lungu, M. (2009). “Visualizing Co-Change Information with the Evolution Radar”, In IEEE Trans. Softw. Eng. 35, 5 (September 2009), 720-735.

Diehl, S. (2007). “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, Springer-Verlag New York, Inc.

Eick, S. G., Steffen, J. L., and Sumner, E. E. Jr. (1992). “Seesoft-A Tool for Visualizing Line Oriented Software Statistics”, In IEEE Trans. Softw. Eng. 18, 11 (November 1992), 957-968.

Lanza, M. (2001). “The evolution matrix: recovering software evolution using software visualization techniques”, In Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01). ACM, New York, NY, USA, 37-42.

Lanza, M. and Ducasse, S. (2003). Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. IEEE Trans. Softw. Eng. 29, 9, 782-795.

Novais, R. L. ; Lima, C. A. N. ; Carneiro, G. F. ; Simoes Junior, P. R. M. ; Mendonça Neto, M. G. (2011b). “An Interactive Differential and Temporal Approach to Visually Analyze Software Evolution”, In: 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis, Williamsburg. VISSOFT11, v. 1. p. 88-91.

Novais, R. L., Carneiro, G. F., Simoes Junior, P. R. M., Mendonça Neto, M. G. (2011a). “On the Use of Software Visualization to Analyze Software Evolution - An Interactive Differential Approach”, In: 13th International Conference on Enterprise Information System, Beijing. v. 3. p. 15-24.

Novais, R., Nunes, C., Lima, C., Cirilo, E., Dantas, F., Garcia, A., Mendonça, M. (2012). “On the Proactive and Interactive Visualization for Feature Evolution Comprehension: An Industrial Investigation”, Proceedings of ICSE, SE in Practice. Zurich, Switzerland. IEEE, pp 1044-1053.

Robillard, M., Weigand-Warr, F. (2005). “Concernmapper: simple view- based separation of scattered concerns”, Proc. of the OOPSLA workshop on Eclipse Technology, ACM, pp. 65–69.

Shneiderman, B. (1996). “The eyes have it: A task by data type taxonomy for information visualizations”, In VL ’96: Proceedings of the 1996 IEEE Symposium on Visual Languages, page 336,Washington, DC, USA. IEEE Computer Society.

Storey, M. D., Čubranić, D., and German, D. M. (2005). “On the use of visualization to support awareness of human activities in software development: a survey and a framework”, In Proceedings of the 2005 ACM Symposium on Software Visualization (St. Louis, Missouri, May 14 - 15, 2005). SoftVis '05. ACM, New York, NY, 193-202.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., Wesslén, A. (2000). “Experimentation in Software Engineering: An Introduction”, Kluwer Academic Publishers, Norwell, MA, USA.

SkyscrapAR: An Augmented Reality Visualization for Software Evolution

Rodrigo Souza[1], Bruno Silva[1], Thiago Mendes[1,2], Manoel Mendonça[1]

[1]Computer Science Department, Federal University of Bahia (UFBA), Bahia, Brazil

[2]Information Technology Dept, Federal Institute of Bahia - Santo Amaro, Bahia, Brazil

{rodrigo, brunocs, thiagomendes, mgmendonca}@dcc.ufba.br

Abstract. *Many software visualizations have been proposed to help stakeholders identify potential threats in the design of evolving software systems, often employing metaphors such as trees, graphs, constellations, maps, and cities, among others. In this scenario, three-dimensional visualizations have also been adopted. They can represent visual metaphors realistically and efficiently engage users. However, the interaction with such visualizations is either cumbersome or requires expensive equipment. This paper presents SkyscrapAR, an augmented reality visualization that employs the city metaphor to represent evolving software, along with its potential applications in the practice. The use of augmented reality allows the user to interact with the city in an intuitive way, by manipulating a marker, while requiring nothing but a computer and an inexpensive camera to make it work.*

1. Introduction

Software systems evolve by nature [Lehman 1980]. The activity of software development becomes more complex as the system evolves, which leads to the need for preventive and corrective maintenance. This complexity is reflected in the fact that about 90% of the total costs of software systems are associated with their maintenance. Therefore, the area represents a promising field for improvement [Erlikh 2000].

Visualization techniques have been used in software engineering as a viable solution to the difficult task of understanding, maintaining and evolving software systems. The visual metaphors are very useful in this context because sight is the most developed sense in human being [Diehl 2007].

During the last decade, tools such as SeeSoft and CodeCity have used 3D visualizations to represent, and hopefully provide a better understanding of software. 3D visualization is sometimes more intuitive and adds an extra dimension that allows the visualization of a larger amounts of information in relation to 2D visualization [Teyseyre 2009]. However, it also has an important limitation. The computer screen is bi-dimensional and always occludes something that is represented in three dimensions. This requires the user to continually interact with the visual scenarios to rotate the visual scene. Current interface devices, such the computer mouse, make it cumbersome for users to manipulate the six degrees of freedom (three translational axes and three rotational axes) of a 3D scene.

A solution to the problem of interaction between the user and 3D software is the use of augmented reality. This technology makes virtual objects visible in real environments, facilitating virtual interaction with the user, since the communication can be performed through gestures and movements [Azuma 1997].

This paper presents SkyscrapAR, a new software evolution visualization approach that uses a city graphical metaphor combined with augmented reality techniques. SkyscrapAR represents software classes and packages in a particular project revision as 3D buildings. These buildings are laid over on a terrain baseline blueprint derived from all versions of the software project. Buildings are sized up and colored by their size and number of changes. The shown project revision can be dynamically chosen and buildings can be dynamically filtered for better evolution analysis. The augmented reality resource allows for easy manipulation of the produced visual scenarios. To our best knowledge, there is no other work in the literature that uses this type of approach.

This paper is structured as follows. Section 2 introduces relevant related work. Section 3 presents SkyscrapAR and its features. Then, Section 4 discusses some applications of the tool. Finally, Section 5 concludes this article.

2. Related Work

According to Steinbrückner and Lewerentz (2010) and Teyseyre (2009), several tools have been proposed in the literature that use 3D visualizations to support software maintenance and evolution activities.

Without implementing it, Parnas (2003) proposed a 3D approach that depicts software production cost related program information to support software maintenance. The idea is to help maintainers and managers to decide which components in a system are superfluous, cause high cost due to frequent changes, and needs to be revised.

Wettel and Lanza (2008) introduced an approach called CodeCity, which can help to detect software design problems by presenting software in the form of a 3D city where the artifacts are mapped to the city's neighborhoods and buildings. Buildings represent classes sized accordingly to metrics such as number of methods and number of attributes. SkyscrapAR is inspired by CodeCity, but it uses different metrics, such as lines of code and code churn (the total number of lines changed in the evolving classes).

Maletic et al. (2001) proposed a virtual reality environment, called Imsovision, to investigate how immersive visualization environments can assist in software development and maintenance. Their environment requires CAVE, an expensive virtual reality system composed of projectors, special glasses, and an electromagnetic tracking system. Although SkyscrapAR does not create the same level of immersion, it provides an intuitive way to manipulate 3D models while requiring only an inexpensive camera.

3. SkyscrapAR

SkyscrapAR is an augmented reality (AR) software evolution visualization based on the metaphor of an evolving city, see Figure 1. Similarly to CodeCity, SkyscrapAR represents packages (or folders in the source code file structure) as rectangular city lots, with sub-packages being stacked on top of the package containing them. Classes (or source code implementation files) are represented by buildings (boxes with different areas and heights) located on top of their respective packages, with the area they occupy being proportional to the size of the class, measured in lines of code. The user can browse over revisions of the software, and see buildings appearing, disappearing, and changing their shape to reflect successive modifications that they suffered over time.

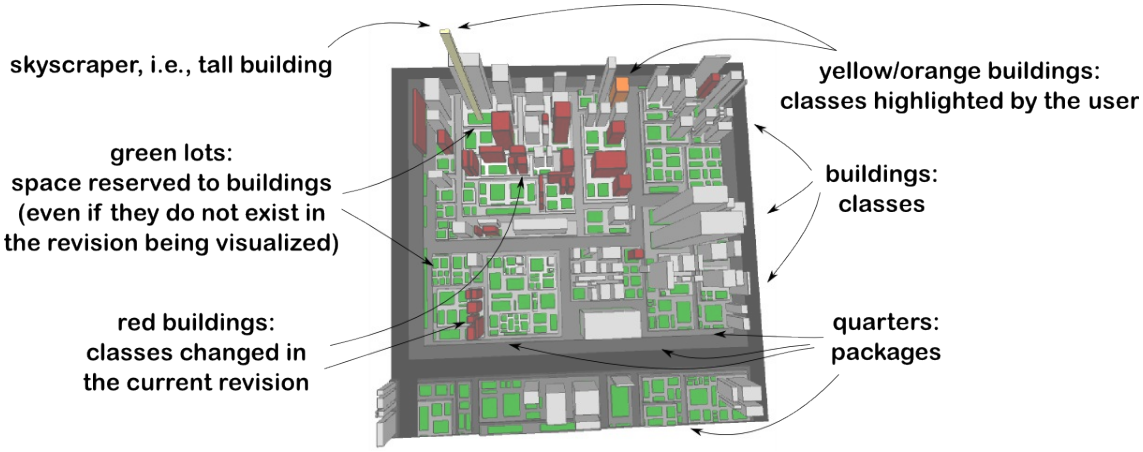


Figure 1. The JUnit framework visualized as a city using SkyscrapAR.

Because the city (i.e., the software) evolves over time, it must have space to accommodate all buildings that existed in some period of its history. That is why the city terrain is divided in green lots, each one belonging to a building (i.e., a class). Each green lot is large enough to fit its respective building when it reaches maximum area. So, for example, buildings which present some green area in the last revision represent classes that used to be larger but had their size reduced.

The city visualization is projected onto the user's real-world environment, as captured by a camera. The user interacts with the visualization as he would do with a mock-up city, manipulating it with the hands so it can be seen from the desired viewpoint.

A demonstration video[1] of the tool can be found online. Readers are urged to look at it before reading further on. The full source code[2] is also publicly available under an open source license. The rest of this section describes SkyscrapAR in terms of its architecture, used source code metrics, and its user interface.

Architecture

SkyscrapAR is composed of two executables: the extractor and the viewer. The extractor, written in Java, reads a local Git repository containing Java source code and outputs a XML file describing its revisions, along with information for each source code file that was modified in each revision. The viewer, written in Processing, takes the XML file and presents its data as a city that the user can view and interact with.

The hardware setup needed to use the visualization consists of a computer with a camera and a printed marker. A marker is a piece of paper or any other object with a predefined black and white square pattern printed on it, as shown in Figure 2a. The marker signals where the city should appear within the user's environment.

Each frame captured by the camera is converted to a binary, black and white image, according to a configurable brightness threshold. After that, an algorithm detects black square contours and then compares its inner region with a predefined pattern. If they match, then it builds a local 3D coordinate system for the marker, as shown in Figure 2b. The 3D model of the city is then aligned with the local coordinate axes, as shown in Figure 2c, so it moves together with the marker, giving the effect that it is a part of the real world.

[1] Available at http://www.youtube.com/watch?v=VVRjihr-40U.

[2] Available at https://github.com/rodrigorgs/SkyscrapAR.

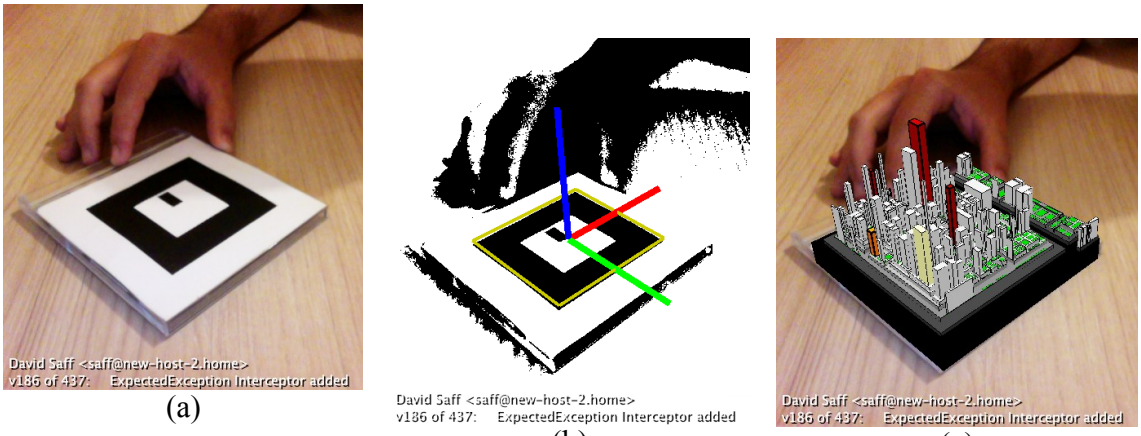


Figure 2. Augmented reality: (a) the camera captures a scene containing a marker; (b) after the image is transformed to black and white, the marker is detected and its local coordinate system is computed; (c) the 3D model is aligned with the coordinate axes.

Mapping Metrics to Visual Elements

A software system is composed of consecutive revisions. In each revision, two metrics are computed for all the classes: number of lines of code (LOC) and code churn.

The LOC metric is the total number of lines in the respective source code file, including comments and blank lines. This metric, although simple to measure, correlates with many traditional complexity metrics that are used to predict development effort and fault proneness [El Elmam et al. 2001].

Code churn refers to how much a particular file has been modified in its current and previous revisions in the version control system. The code churn of a class in its initial revision is equal to its LOC. Then, each time the class is modified, the code churn is incremented by the number of lines of code added or removed in the revision, as computed by Unix's *diff* tool. That way, code churn always increases or is kept constant over time. The rationale for code churn is that source code files modified frequently are less stable and tend to be more prone to faults [Nagappan and Ball 2005].

The two metrics are mapped to visual attributes of the visualization. The LOC metric for a class determines the area of the base of the respective building, so that larger classes are represented by buildings that occupy a greater part of the city's territory.

All buildings start with a fixed, minimum height. In the subsequent revisions, the height of a building is determined by how much the code churn of its respective class increased since its initial revision. Therefore, classes that undergone many changes since their creation are represented by tall buildings, or *skyscrapers*.

It should be noted that code churn, unlike LOC, is a historical metric, i.e., its value for one class in one revision reflects the evolution of the class in the previous revisions. Its use in the visualization, therefore, enables the user to gather historical information about the software by looking at a single static view.

User Interface

The user interface for SkyscrapAR displays the scene being captured by the camera and, if a marker is on the scene, the 3D model of the city is superimposed on it. In the bottom

part of the screen, information about the current revision being visualized is displayed. It includes the sequential number of the revision, together with the name of the developer responsible for the change and the message describing the change.

Contrary to what happens in most 3D software visualizations, in which users change their viewpoint by using the mouse or the keyboard, in SkyscrapAR, the user can view the city from different angles by manipulating either the 3D marker or the camera. With such interaction mechanism, the user can explore all six degrees of freedom of the 3D space (translation and rotation along three axes) in a natural way. For instance, the city can be seen from the top, in an aerial view, evidencing the structure of the software, or sideways, in a skyline view, evidencing skyscrapers, i.e., classes with high code churn.

In its current state, SkyscrapAR still depends on mouse and keyboard input, although we plan to reduce this dependency. The mouse pointer is used to highlight classes the user may want to focus on. The keyboard is used to navigate through revisions (left and right arrow keys), filter out buildings ("H" key), zoom in ("Z" key), and zoom out ("Shift-Z" key combination).

When the user advances the visualization to the next revision, buildings representing classes that were modified in that revision appear in red, so they can be easily spotted among the default gray buildings. It is expected that the buildings in red changed its shape compared to the previous revision[1] to reflect changes in the number of lines of code (area of the base) or in its code churn (height). Such changes in shape are presented as a smooth animation, by interpolating the dimensions of the buildings over a short period of time (less than one second). That way, it is easier to track the changes in the classes over time, enhancing the perception of evolution.

When the mouse pointer is over a building, the interface shows details on demand for the respective class: name, package containing it, and the values for the LOC and code churn metrics. The user can also highlight a set of buildings, which get painted in yellow, by clicking on them. If a class is highlighted and has changed in the current revision, it gets painted in orange instead (i.e., a red and yellow mix).

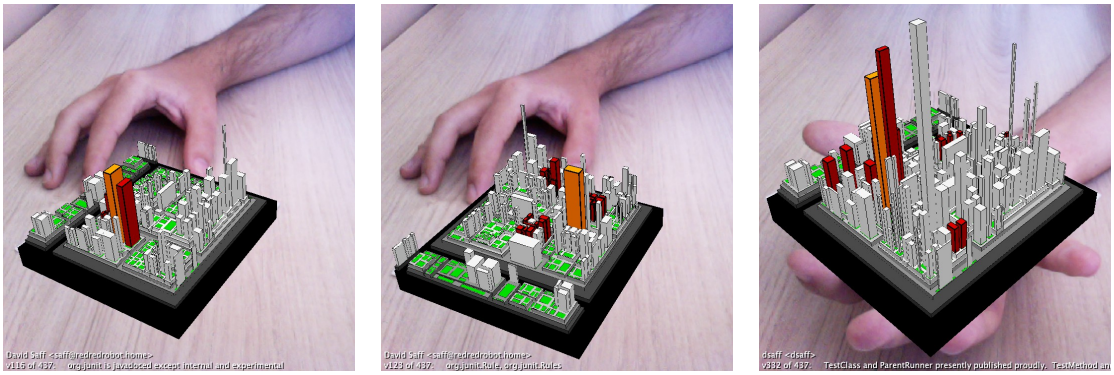
The purpose of the highlighting is twofold: it helps the user track the position of some buildings while exploring the visualization, and enables the user to focus on the changes of the highlighted buildings. When at least one building is selected, the navigation among revisions is restricted to revisions in which at least one highlighted class is changed. This is useful to study the evolution of a set of classes.

By pressing the "H" key on the keyboard, gray buildings are filtered out, i.e., only highlighted classes and classes that were changed in the current revision are displayed. This filter helps the user to analyze the scattering of changes throughout the packages. When combined with the restricted navigation that takes place when classes are highlighted, it also enables the user to visualize classes that evolve together within a given set of classes.

4. Applications in Software Development Practice

Normally, developers spend more time understanding the code than modifying it [LaToza 2006], partly because, as time goes by, software systems get harder to understand and to maintain unless effort is invested to simplify the design [Lehman 1980]. Code review meetings and module inspections are examples of team effort for identifying design flaws that could be mitigated to improve the software comprehension and maintenance.

[1] One notable exception is the case when only the file permissions or ownership were changed, although its contents were kept unchanged.



(a) (b) (c)

Figure 3. Three distinct versions of the JUnit framework in which the class BlockJUnit4ClassRunner (highlighted in orange) was modified.

Moreover, researchers have found demand for *agile assessments*, that is, a kind of software product assessment that may provide valuable information rapidly and cheaply to support program understanding and maintenance [Nierstrasz 2012]. We envision several applications of SkycrapAR in this context. This section describes some of them and shows screenshots of the tool to illustrate typical situations (see Figure 3).

Finding Skyscrapers

In a software project, it is common to find classes which change a lot—the so called change-prone classes. In SkyscrapAR, they can be easily found as *skyscrapers* (see the orange building in Figure 3). Analog to the real concept, skyscrapers in our visualization are classes that concentrate a lot of the development effort. These classes deserve special attention, since they are more likely to be fault-prone [Nagappan and Ball 2005].

Visualizing Scattering of Changes

Source code management systems usually list packages and classes that were changed in each revision, but there is no visual information about the scattering of each commit transaction that generates a revision. This is easily done in SkycrapAR (see red buildings in Figure 3). Identifying scattered changes is important, as they represent modifications that impacted several systems modules, pointing out system-wide maintenance activities, such as refactorings or move operations among folders, or indicating that there are too many dependencies between software modules. By focusing on this second issue, developers can detect candidates for the Shotgun Surgery, identifying design flaws that cause modifications in one module and trigger modifications in several other modules.

Visualizing Classes Co-change

Sometimes, classes have logical couplings even if they are not explicitly coupled to each other by their internal structural elements [Gall 1998]. Classes may be connected by means of implicit abstract elements related to the domain. For example, classes that are not structurally coupled but implement concerns that are strongly dependent at the requirements level. SkycrapAR provides information about the classes co-change by highlighting in red the classes that changed together in each revision (see Figure 3).

Finding God Classes

A god class is a well-known code smell which refers to classes that perform too much work on its own. It breaks one of the basic principles of object-oriented design which

states that a class should have one single responsibility. Also, god classes are more likely to undergo changes as much as the evolution history targets the responsibilities implemented by the class [Silva 2012]. SkyscrapAR helps developers reason about god classes using the *High Occupation of Territory* metaphor—tall buildings occupying large areas. An analogy can be made to a real city. While, in real world, buildings that occupy a larger area require their dwellers to pay higher taxes, in the visualization it is the project developers who have to pay the price of maintaining large classes.

Identifying Populous Districts and City Centers

Normally *systems as cities* have one or more neighborhoods with tall buildings and packages with no or very few green area—those areas are *Populous Districts*. This observation can be interpreted as the most touched parts of the system. If there is only a single well-defined populous district, we call it the *City Center*. Finding the city center (or other populous districts) may be useful to focus code inspection and testing on the most modified and possibly the most fault-prone parts of the system. It may also help to find refactoring opportunities on specific packages that suffered too many changes.

Identifying Rural Houses

Rural houses are classes that lost their functionality along with the software evolution, so they are now presented as large green areas with a small building in the last revision of the software. Classes like that should be examined for signs of dead code, i.e. they may be obsolete and therefore be removed.

5. Final Remarks

This paper presented SkyscrapAR, an augmented reality visualization that employs the city metaphor to display information about the evolution of a software system. To the best of our knowledge, this is the first software visualization tool to employ augmented reality. In addition to describing SkyscrapAR, the paper shows how it can be used to support software comprehension tasks.

Currently, the tool has some limitations. In its current stage, only Java source code is supported, although it should be easy to add support for other languages. Also, it is limited to display one software system at a time, even if multiple markers are used. Finally, the user interaction still relies on the use of mouse and keyboard, which reduces the sense of immersion that characterizes augmented reality applications.

As future work, we intend to investigate interaction modes that dispense the use of mouse and keyboard. Our goal is to create a user interface that is intuitive, while discarding approaches that require expensive equipment such as virtual reality helmets and gloves. We also plan to make better use of colors in the visualization in order to display richer information. For instance, buildings can be painted in different colors to represent distinct crosscutting concerns implemented by them or, still, the developers who have contributed to the class. We plan to evaluate the tool with developers in academic or industrial settings. The evaluation should focus on usability and the effectiveness of the tool when used to support specific software comprehension tasks.

Acknowledgements

The authors wish to thank Michele Lanza for encouraging this publication.

This work was partially supported by CNPq/INES under grant 573964/2008-4.

References

Azuma, R. (1997) “A Survey of Augmented Reality”. Presence: Teleoperators and Virtual Environments, v. 6, n. 4, August, p. 355–385.

Diehl, S. (2007) “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, Springer-Verlag New York, Inc.

El Emam, K., Benlarbi, S., Goel N., and Rai S. N. (2001) “The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics”. IEEE Transasctions on Software Engineering, 27(7).

Erlikh, L.(2000) “Leveraging legacy system dollars for e-business”. IEEE IT Pro, pp. 17–23.

Gall, H., Hajek, K. and Jazayeri, M.. (1998) “Detection of Logical Coupling Based on Product Release History”. In Proceedings of the International Conference on Software Maintenance (ICSM '98). IEEE Computer Society, Washington, DC, USA.

Lanza, M., Marinescu, R, and Ducasse, S. (2005) “Object-Oriented Metrics in Practice”. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

LaToza, T., Venolia, G. and DeLine, R.. (2006) “Maintaining mental models: a study of developer work habits". In ICSE '06: Proceedings of the 28th international conference on Software engineering, pages 492–501, New York, NY, USA.

Lehman, M. M., (1980) "Program Life Cycles and Laws of Software Evolution," Proceedings of IEEE, Special Issue on Software Engineering, September, pp. 1060–1076.

Maletic J. I., Marcus A., Dunlap G., and Leigh J., (2001) “Visualizing Object-Oriented Software in Virtual Reality,” Proc. Ninth Int’l Workshop Program Comprehension (IWPC ’01), p. 26.

Nagappan, N., and Ball, T., (2005) "Use of Relative Code Churn Measures to Predict System Defect Density", Proc. the 27th International Conference on Software Engineering (ICSE '05), St. Louis, MO, USA, May 15-21, 2005, pp. 284–292.

Nierstrasz, O. (2012) “Agile software assessment with Moose”. SIGSOFT Softw. Eng. Notes 37, 3 (May 2012), 1-5.

Panas, T., Berrigan, R. and Grundy, J. (2003) “A 3D metaphor for software production visualization”. International Conference on Information Visualization, page 314.

Silva, B., Sant’Anna C., Chavez, C. and Garcia, A. (2012) “Concern-based Cohesion: Unveiling a Hidden Dimension of Cohesion Measurement”. In: 20th IEEE International Conference on Program Comprehension, ICPC’12, Passau, Germany.

Steinbrückner, Frank, Lewerentz, Claus (2010) “Representing development history in software cities”. SoftVis '10, Proceedings of the 5th international symposium on software visualization. ACM, NY, USA.

Teyseyre, A. R., Campo, M. R., (2009) "An Overview of 3D Software Visualization", IEEE Transactions on Visualization and Computer Graphics, pp. 87–105.

Aplicação de uma técnica de visualização de dados baseado em árvores para auxiliar a priorização de requisitos em projetos ágeis

Fábio Abrantes Diniz[1], Thiago Reis da Silva[1], Íthalo Bruno Grigório de Moura[1], Diego Grosmann[1], Francisco Milton Mendes Neto[1], Pedro Fernandes Ribeiro Neto[1]

[1]Programa de Pós-Graduação em Ciência da Computação – PPgCC
Universidade do Estado do Rio Grande do Norte – UERN
Universidade Federal Rural do Semiárido – UFERSA
BR 110 – Km 46 – Bairro Costa e Silva – Campus Central
59.625-620 Mossoró – RN, Brasil

{fabio.abrantes.diniz, trsilva.si, ithalobgm, diegogrosmann}@gmail.com, miltonmendes@ufersa.edu.br, pedrofernandes@uern.br

Abstract. *The practice of prioritizing requirements involves analysis of the value of each requirement on the part of clients and selection of requirements that will be implemented in a particular version of the system. Wrong choices during the prioritization of requirements can affect the quality of the system, and consequently its acceptance by customers. Inserted in this context, this paper proposes a technique for data visualization, map-based tree (TreeMaps), to assist in visualizing the results of a practice that uses the technique of Kano to prioritize design requirements that address the agile Scrum methodology. For the results of this practice are difficult to be analyzed to determine the real importance of each requirement.*

Resumo. *A prática da priorização de requisitos envolve a análise da valorização de cada requisito por parte dos clientes e seleção dos requisitos que irão ser implementados em determinada versão do sistema. Escolhas erradas durante a priorização dos requisitos pode afetar a qualidade do sistema, e consequentemente sua aceitação pelos clientes. Inserido neste contexto, este artigo propõe uma técnica de visualização de dados, baseado em mapa de árvores (Treemaps), para auxiliar na visualização dos resultados de uma prática que utiliza a técnica de Kano para priorizar requisitos de projetos ágeis que abordam a metodologia Scrum. Pois, os resultados dessa prática são difíceis de serem analisados para determinar a real importância de cada requisito.*

1. Introdução

A priorização de requisitos é uma atividade desafiadora do desenvolvimento de software [Lamsweerde 2000]. Esta prática envolve a análise de cada requisito por parte dos *Stakeholders* (partes interessadas) e seleção dos requisitos que irão ser desenvolvidos em determinada versão do sistema. Nas metodologias ágeis, algumas práticas de priorização de requisitos têm sido adotadas em projetos ágeis para melhorar a etapa de

priorização dos requisitos. Entre elas, encontra-se uma prática de priorização de requisitos baseada na técnica de *Kano*, que foi aplicada em desenvolvimento de software que aborda a metodologia *Scrum* [Asfora 2009].

A técnica de *Kano* possibilita aos desenvolvedores de produtos transformarem as informações obtidas pelas pesquisas em melhorias reais no produto de forma a buscar a satisfação do cliente [Kano 1984]. No entanto, a técnica de *Kano* gera muita informação e a apresentação textual da mesma, por meio de tabelas e relatórios, faz com que os responsáveis gastem muito tempo processando grandes volumes de dados.

Portanto, este artigo utilizou uma técnica de visualização de dados baseada em Mapa de Árvore, conhecido como "*TreeMap*" [Jonhnson e Shneiderman 1991], com o objetivo de mostrar novas formas de visualização de dados, mais intuitivas e eficazes, para auxiliar a análise dos resultados originados da prática de priorização de requisitos baseada na técnica de *Kano*.

Este artigo encontra-se organizado da seguinte forma: na Seção 2 apresenta uma breve descrição da metodologia *Scrum*, detalhando as suas características usadas no gerenciamento dos projetos de software. Na Seção 3 descreve a prática adotada para a priorização de requisitos baseada na técnica de *Kano* e a sua adaptação para ambientes ágeis que utiliza a metodologia *Scrum*. Na Seção 4 aborda a técnica de visualização utilizada neste artigo, conhecida como *TreeMap*, para auxiliar a prática de priorização de requisitos em projetos ágeis, dando suporte na visualização dos dados gerados. Na Seção 5 é apresentado um protótipo chamado *TreeSolutions* para testar a técnica *TreeMap* e os seus resultados. As conclusões finais e trabalhos futuros são apresentados na Seção 6.

2. Scrum – Uma Metodologia de Desenvolvimento Ágil de Software

A metodologia ágil *Scrum* surgiu com a proposta de "desburocratizar" o processo de desenvolvimento de software, permitindo que as equipes sejam mais adaptáveis, respondendo rapidamente às constantes mudanças nos projetos de software. De acordo com seus evangelizadores, o cliente fica mais satisfeito, pois constantemente há entrega de funcionalidades desenvolvidas, e ele participa ativamente no projeto, trazendo seu conhecimento sobre o próprio negócio.

O *Scrum* se destaca dos outros processos ágeis por ser um método iterativo, incremental e ágil para o gerenciamento de Projetos [Schwaber 2004]. O processo inicia com os requisitos do projeto organizados em uma lista de requisitos, chamada de *Product Backlog,* em ordem decrescente de prioridade. A equipe separa uma parte do topo do *Backlog* para o *Sprint*, formando o *Sprint Backlog* (lista de tarefas do *Sprint*). O *Scrum* trabalha com desenvolvimento incremental, onde cada iteração é chamada de *Sprint*. Os *Sprints* são curtos, tendo duração de 30 dias.

Segundo [Schwaber 2004], durante um *Sprint,* a equipe tem autonomia para decidir como as tarefas serão implementadas e garante que os requisitos mais importantes sejam desenvolvidos primeiro. Além disso, a equipe tem curtas reuniões diárias, sempre no mesmo horário, junto com o *Scrum Master* (responsável pela gestão do projeto e liderança do grupo), chamadas de *Scrums*. Nessas reuniões é discutido o

andamento do trabalho, onde cada membro da equipe responde às questões: o que fiz desde ontem? O que pretendo fazer até amanhã?

A saída do *Sprint* é um conjunto de funcionalidades 100% desenvolvidas, que serão aprovadas pelo *Product Owner* (responsável pela definição do projeto, priorização dos requisitos e definição dos mesmos) e entregues ao cliente. Ao final de cada iteração, toda a equipe participa de uma retrospectiva do *Sprint.* Após a conclusão do *Sprint*, reinicia-se o ciclo, retirando-se a próxima fatia do *Product Backlog* para o próximo *Sprint* [Lee e Newcomb 1997].

3. A Técnica de *Kano* na Priorização de Requisitos para Projetos Ágeis

A aplicação da técnica de *Kano* no processo de priorização para projetos ágeis baseadas na metodologia *Scrum* é realizada antes do *Product Owner* enviar o *Product Backlog* para a equipe de desenvolvimento, a fim de que os requisitos primeiro possam ser priorizados. A técnica consiste em fazer um par de perguntas para cada requisito ao cliente. Uma pergunta de inclusão de um específico requisito e outra de exclusão deste específico requisito. Segundo [Asfora 2009], a importância da técnica de *Kano* em aplicar essas perguntas é mostrar o paralelo realizado entre o que deixa o cliente muito satisfeito e o que o deixa muito insatisfeito para sem ter a real noção de necessidade de cada requisito.

As perguntas são formadas da seguinte forma: (a) Como você se sentiria caso o Requisito X estivesse no próximo *release*? e (b) Como você se sentiria caso o Requisito X NÃO estivesse no próximo *release*?. As opções de respostas estão especificadas na Tabela 1.

Tabela 1: Respostas às perguntas de inclusão e exclusão do requisito e resultados para cada combinação [Asfora 2009].

		Perguntas de Exclusão do Requisito				
Perguntas de inclusão do requisito		Muito Satisfeito	Satisfeito	Pouco Satisfeito	Indiferente	Insatisfeito
	Muito Satisfeito	Q	D	D	D	L
	Satisfeito	R	I	I	I	M
	Pouco Satisfeito	R	I	I	I	M
	Indiferente	R	I	I	I	M
	Insatisfeito	R	R	R	R	Q

A categoria I (Indiferente) na Tabela 1 é utilizada quando o usuário demonstra que não tem necessidade real para esta funcionalidade, ou seja, tanto faz se ela é satisfeita ou não. A categoria M (Mandatório ou Indispensável) significa que se o requisito não forem feitos o cliente fica muito insatisfeito. Para o cliente, esses requisitos já estão embutidos nos produtos oferecidos, sendo, portanto, um pré-requisito. O fato de colocar os requisitos Mandatórios não tornará o cliente mais satisfeito, no entanto, sem eles o sistema não funciona e o cliente não adquire o produto. A categoria D (Desejado) significa que esses requisitos são requisitos que proporcionam grande satisfação ao cliente quando estão presentes, porém não representam insatisfação caso

não estejam presentes. A categoria L (Linear ou Importante) significa que a satisfação do cliente é proporcional ao nível de preenchimento desses requisitos, ou seja, quanto maior o nível de preenchimento, maior será a satisfação do cliente e vice-versa. A categoria R (Reverso) significa que, se aquele requisito for desenvolvido, poderá trazer uma rejeição ao software ou a determinada funcionalidade. A categoria Q (Questionável) significa que o usuário não entendeu as perguntas ou que ele não está correspondendo com a verdade [Asfora 2009]. De acordo com a análise de [Asfora 2009] mostra que a ordem de priorização dos requisitos que trariam um valor muito grande se forem atendidos é respectivamente: Mandatórios, depois os Importantes ou Lineares e, por fim, os Desejáveis.

De acordo com a Tabela 2, a combinação das respostas da tabela de *Kano* para as duas perguntas gera um resultado para cada requisito. Podemos observar que o requisito 1 (Req 1) possui a maior percentagem (43,8%) para classificação como Linear ou Importante. O requisito 3 (Req 3) foi classificado como Mandatório e Desejado, pois ambos possuem resultados próximos. Logo, deixando a seguinte ordem decrescente de priorização: Requisito 2, Requisito 1 e finalmente Requisito 3.

Tabela 2: Sumarização dos resultados.

Tema	D	L	M	I	R	Q	Classificação
Req 1	18,4	**43,8**	22,8	12,8	1,7	0,5	**Linear**
Req 2	8,3	30,9	**54,3**	4,2	1,4	0,9	**Mandatório**
Req 3	**39,1**	14,8	36,6	8,2	0,2	0,1	**Desejado Mandatório**

4. *TreeMap* – Uma Técnica de Visualização de Dados baseada em Árvore

TreeMap é uma técnica de visualização que explora conceitos básicos de ergonomia, fazendo com que o ser humano foque inicialmente seu olhar em figuras grandes para depois olhar para figuras pequenas [Jonhnson e Shneiderman 1991]. Essa característica pode ser utilizada na visualização dos itens do *Product Backlog*, com o objetivo de exibir a hierarquia dos requisitos que têm mais prioridade de serem desenvolvidos.

Algumas vantagens são evidentes na utilização *TreeMap*, tais como: utiliza eficientemente toda a tela de visualização; preserva o contexto geral da informação; navegação rápida entre os nós; permite ao usuário a visão geral do escopo e é muito útil na exibição de variáveis quantitativas dos dados [Jonhnson e Shneiderman 1991].

O *TreeMap* divide a tela de exibição em uma sequência aninhada de retângulos correspondentes a atributos de um conjunto de dados. Cada retângulo possui área e cor as quais são definidas por valores previamente estipulados. Logo possui propriedades que devem ser levadas em consideração e que, de acordo com [Jonhnson e Shneiderman 1991], são:

1. Um peso (classificação do requisito) determinará o tamanho de cada retângulo na estrutura;
2. A cor representará a satisfação do usuário em ter o requisito no sistema [Kano 1984]. As cores mais claras representarão os requisitos mais desejados pelos usuários, enquanto as cores mais escuras representarão os requisitos menos

desejados. Este artigo implementa uma escala de cores que atribui valores únicos para cada cor que os diferenciam;

3. Uso de *pop-ups* exibe informações do requisito quando o *mouse* é passado sobre a região do retângulo de interesse.

Outro ponto forte na construção do *TreeMap* é a definição do algoritmo a ser utilizado na criação dos retângulos. Os *layouts* dos retângulos dependem do algoritmo de divisão utilizado. Alguns algoritmos utilizados no *TreeSoltuions* foram: *Slice and Dice, Squarified TreeMap* e *Ordered TreeMap* [Jonhnson e Shneiderman 1991]. Estes algoritmos representam uma estrutura de dados hierárquica por divisão recursiva de retângulos. Essa recursão determina o *layout*, calculando a área do retângulo e preenche os retângulos em suas respectivas localidades.

5. TreeSolutions

Com base nos conceitos dos *TreeMap*, o protótipo *TreeSolutions* foi projetado para atender o *Product Owner* na etapa de priorização de requisitos da metodologia *Scrum* que utiliza a técnica de *Kano*. Ajudando o *Product Owner* nas tomadas de decisões na determinação do valor de negócio dos requisitos do *Product Backlog* a fim de que possa ser entregue à equipe de desenvolvedores mais rapidamente.

O protótipo é *open source* e desenvolvido na linguagem *Java*. Foram feitos testes com os dados presentes nos estudos de caso encontrados na dissertação elaborado por [Asfora 2009], e de acordo com a análise dos resultados observou-se que o uso da técnica *TreeMap* tornou a visualização dos dados mais eficiente. Nas subseções seguintes, é descrito o protótipo *TreeSolutions* com suas funcionalidades e o resultado obtido pelo *TreeMap* com os dados do estudo de caso da dissertação de [Asfora 2009].

5.1. Implementação do Protótipo TreeSolutions

O *TreeSolutions* consiste em um protótipo *desktop* desenvolvido na linguagem de programação *Java* para testar a técnica *TreeMap* nos dados vindos da tabela de *Kano*. Este protótipo está subdividido em dois módulos principais de implementação: Módulo *TreeMap* e Módulo Interface.

No Módulo Interface, o *Product Owner* tem acesso a uma interface gráfica com informações sobre seus *Product Backlog* contendo os requisitos. Esta interface possibilita inserção do *Product Backlog* (lista de requisito). Além disso, são inseridas as informações vindas dos resultados da tabela de *Kano*, tais como: os valores percentuais das categorias de todos os requisitos (Mandatório, Linear, Desejado, Questionável, Reverso ou Indiferente). As mesmas podem ser importadas e exportadas em arquivo no formato XML (*eXtensible Markup Language*).

No Módulo *TreeMap*, é implementada a técnica de visualização chamada *TreeMap*. Utilizou-se a classe *TreeMap* de *Java* que oferece funcionalidades adicionais de associar uma ordem aos elementos da coleção. Os algoritmos, citados na Seção 4, foram desenvolvidos e aplicados nesse módulo para representar as áreas dos retângulos e foram implementados os provedores de cores, que importam os modelos de cores RGB (*Red,Green e Blue*), HSB (*Hue, Saturation e Brightness*) e suas variações, para a reprodução das cores nas áreas dos retângulos do *TreeMap.*

De acordo com o contexto da implementação do protótipo, foi dada a área do retângulo como um atributo mais significativo que a cor, pois a área representa requisitos indispensáveis e importantes, ou seja, são requisitos que, sem eles, o sistema não funcionaria. A cor do retângulo representa a satisfação do usuário em ter o requisito no sistema, ou seja, quanto mais clara for a cor na escala das cores, maior é a satisfação do usuário em ter o requisito no sistema. A área do retângulo indica o grau de presença, isto é, retângulos de maior área representam os requisitos que têm que estar presentes no sistema. Logo, quanto maior for a área e maior for a cor na escala das cores, indica que esse requisito tem maior prioridade e deve ser implementado primeiro.

Para calcular o tamanho da área do retângulo, devido à sua função de servir para "medir o grau de presença no protótipo", foi feita uma média ponderada com as entradas dos dados da categoria Mandatório e Linear, com seus respectivos pesos dados como entrada ou por um padrão *default* do sistema escolhido. Os valores *default* são: Mandatório = 10, Linear = 5, desejado = 2, indiferente = 0, questionável = -1, reverso = -2. Estes valores foram atribuídos de acordo com os contextos de suas características e testes analisados durante o desenvolvimento. A categoria Questionável e o Reverso receberam valores negativos, pois categorizam requisitos que não trazem satisfação aos usuários [Asfora 2009].

Além de apresentar ao usuário a visualização dos requisitos em forma de retângulo, o protótipo possui o recurso de *pop-up,* que mostra informações do requisito quando o cursor do *mouse* passa sobre a área que representa um requisito, tais como: o tamanho da área do retângulo, o valor da cor, e os atributos do requisito com seus valores recebidos do resultado da tabela de *Kano*.

5.2. Resultados do Protótipo

O teste do protótipo *TreeSolutions* foi feito em cima dos dados coletados da dissertação de [Asfora 2009]. Os dados adquiridos envolvem um estudo de caso do CESAR (Centro de Estudos e Sistemas Avançados do Recife). Eles foram originados de um projeto do SEPG – Grupo de melhoria do processo interno de desenvolvimento de software. Os dados apresentados na Tabela 3 contêm 15 requisitos com seus resultados vindos da aplicação da técnica de *Kano*.

Tabela 3: Percentual das respostas advindas da técnica de *Kano*. Fonte: [Asfora 2009].

Requisito	Mandatório	Linear	Desejado	Indiferente	Reverso	Questionável
1	0	20	20	40	20	0
2	0	80	0	20	0	0
3	0	60	20	20	0	0
4	0	40	0	60	0	0
5	0	60	20	20	0	0
6	0	20	20	60	0	0
7	0	0	0	100	0	0
8	20	60	0	20	0	0
9	0	0	0	75	25	0
10	40	40	0	20	0	0
11	40	40	0	20	0	0
12	0	20	0	80	0	0
13	20	20	20	40	0	0
14	0	40	40	20	0	0

15	20	40	0	40	0	0

Portanto, a Figura 1 a seguir, ilustra o *TreeMap* resultante da entrada dos dados da Tabela 3. De acordo com a análise dos resultados, observou-se que o uso da técnica *TreeMap* tornou a visualização dos dados mais eficiente e robusta para a elaboração da lista de requisitos priorizados.

Como ilustra a Figura 1, a tela de visualização tem um campo de escolha da estratégia de visualização (a) em que se escolhe o algoritmo usado na construção do *TreeMap* e tem-se a escala de cores dos retângulos que podem ser alteradas conforme o gosto do usuário (d, e). A parte central da visualização (c) ilustra a lista de requisitos do projeto selecionado (f). Por último, quando se posiciona o cursor do *mouse* sobre um dos retângulos, aparece um *pop-up* mostrando informações do requisito (b).

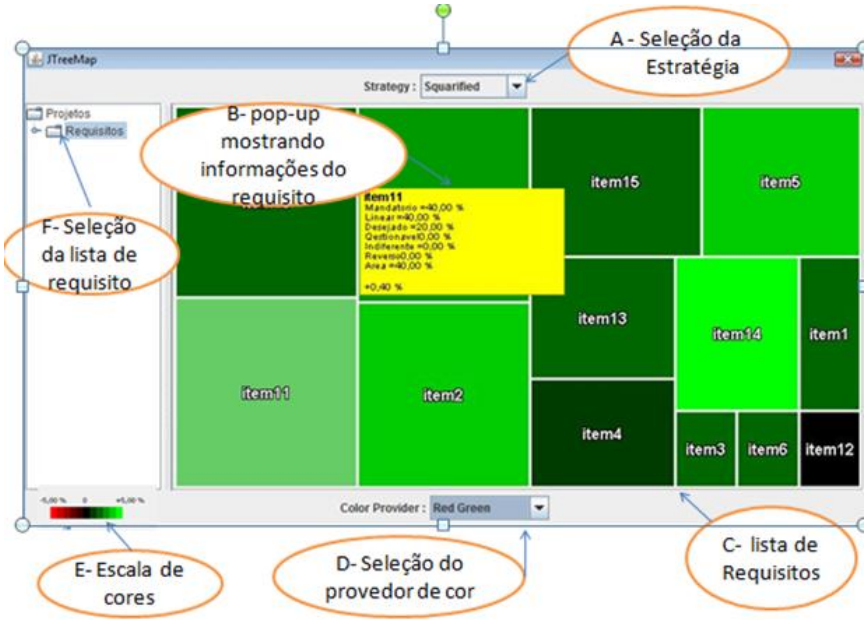


Figura 1: Visualização dos dados da Tabela 3 no *TreeMap*.

A Tabela 4 apresenta a consolidação dos resultados anteriores vindo da análise do *TreeMap,* mostrando tanto a lista de prioridade dos requisitos que devem ser selecionados para serem implementados pelos desenvolvedores quanto os requisitos que devem ficar em espera para futuras versões do software. Os requisitos mais a priori são os que têm menores valores na coluna Prioridade da Tabela 4. É importante lembrar que essa priorização foi formada única e exclusivamente com base na satisfação dos usuários que responderam a pesquisa. É prudente levar em conta dados como esforço e custo para implementar os requisitos antes de gerar a priorização final.

Tabela 4: Tabela de sugestão de priorização.

Prioridades	Selecionados	Em espera
1	Requisito 11	Requisito 1
2	Requisito 10	Requisito 4
3	Requisito 2	Requisito 6
4	Requisito 3	Requisito 7
5	Requisito 5	Requisito 9
6	Requisito 8	Requisito 12

7	Requisito 14	
8	Requisito 13	
9	Requisito 15	

6. Conclusões e Trabalhos Futuros

O objetivo principal deste artigo foi propor uma técnica de visualização utilizando *TreeMap* para ajudar uma prática de priorização de requisitos em projetos ágeis baseado na técnica de *Kano*. Como contribuição também foi desenvolvido um *Applet*, denominado de *TreeSolutions* para aplicação da técnica proposta. Comparado ao processo de visualização dos resultados de forma textuais, percebe-se claramente a melhoria que esta ferramenta traz ao usuário no que diz respeito à tomada de decisão.

O protótipo desenvolvido mostrou-se eficiente em realizar um *feedback* rápido na análise dos dados, gerando menos documentação e menos esforço na priorização dos requisitos. Além de ser simples de usar e fácil de entender. Logo, o *TreeSolutions* auxilia o *Product Owner* a priorizar seu *Product Backlog* mais eficiente e robusta. Consequentemente, as equipes de desenvolvedores podem estimar e desenvolverem os requisitos com mais facilidade.

Como trabalhos futuros pretende-se adicionar o protótipo a tabela de *Kano,* para que possa trazer as entradas de forma mais rápida para o *TreeMap*, gerando as visualizações das informações diretamente da tabela. Pretende-se também implementar outras técnicas de visualização de dados.

Agradecimentos

Os autores agradecem a CAPES pela concessão das bolsas de pesquisa e pelo apoio financeiro para realização da mesma.

Referências Bibliográficas

Asfora, D. M. (2009). “Uma abordagem para a priorização de requisitos em ambientes ágeis”. Dissertação de Mestrado, UFPE.

Johnson, B.; Shneiderman, B. (1991) “*Tree-maps: A spacelling approach to the visualization of hierarchical information structures*”. International IEEE Visualization Conference, v. 1, p. 284-291.

Kano, N. at al. (1984) “*Attractive Quality and Must-be Quality*”. Journal of the Japanese Society for Quality Control. v. 14, n. 14, p. 39-48.

Lamsweerde, A. (2000) “*Requirements Engineering in the year 2000: A Research Perspective*”. 22nd Proceedings of International Conference on Software Engineering. Limerick, Ireland.

Lee, M. C.; Newcomb, J. F. (1997) “*Applying the Kano methodology to meet customer requirements: NASA’s microgravity science program*”. Quality Management Journal, v. 4, n. 3, p. 95-110.

Schwaber, K. (2004) “*Agile Project Management with Scrum*”. Ed. Microsoft Press.

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

	Code Smells				
Programador I	FE	GC	DC	SS	Geral
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.

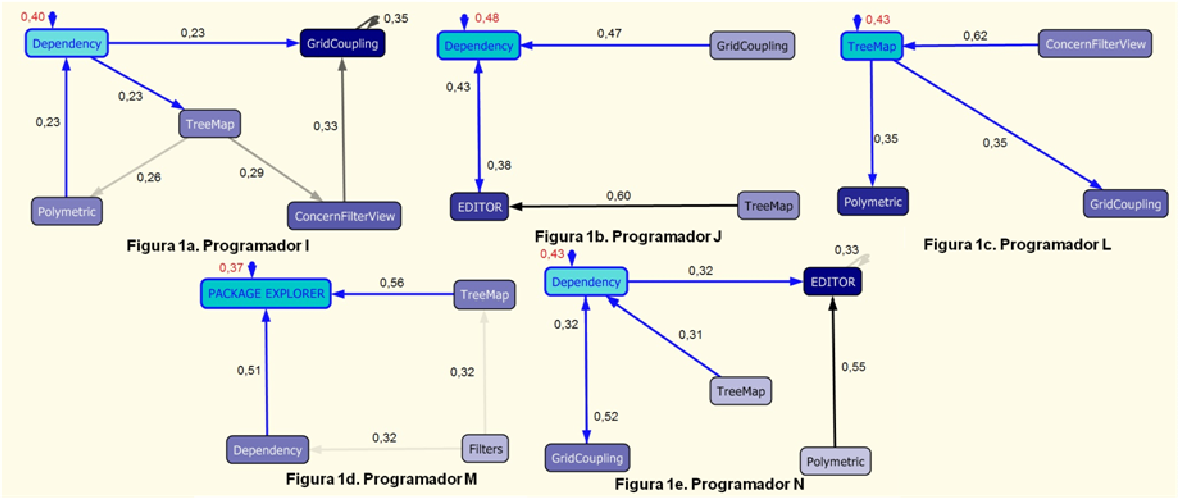


Figura 1. Padrões de Sequência dos programadores

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.

Usando Recursos de Visualização Enriquecidos com Elementos de Percepção para a Compreensão de Software em um Ambiente de Desenvolvimento Distribuído

Carlos F. R. Conceição[1], Glauco de Figueiredo Carneiro[1], José Maria N. David[2]

[1] Departamento de Sistemas e Computação, Universidade Salvador (UNIFACS) - BA

[2] Departamento de Ciência da Computação, Universidade Federal de Juiz de Fora - MG

cfabioramos@unifacs.br, glauco.carneiro@unifacs.br, jose.david@ufjf.edu.br

Abstract. *This paper describes the use of visualization resources enriched by perception elements to enhance software comprehension. For this purpose, the visual environment SourceMiner was extended to provide information that describes actions performed by programmers. The goal is to enable the use of perception elements in a distributed environment. A case study was conducted to analyze the effectiveness of this support to software comprehension activities in a distributed environment. The results provide initial evidences that perception elements enhance software comprehension activities performed by geographically distributed teams.*

Resumo. *Este artigo descreve o uso de recursos de visualização enriquecidos com elementos de percepção para potencializar a compreensão de software. Para esta finalidade, o ambiente visual SourceMiner foi estendido para prover informações que descrevem ações executadas pelos programadores. O objetivo é permitir o uso de elementos de percepção em ambientes distribuídos. Um estudo de caso foi conduzido para analisar a efetividade deste suporte a atividades de compreensão de software em ambientes distribuídos. Os resultados apresentam evidências iniciais de que os elementos de percepção potencializam as atividades de compreensão de software executadas por equipes distribuídas geograficamente.*

1. Introdução

A execução de atividades de compreensão de software em ambientes de desenvolvimento distribuído necessita de uma comunicação eficiente. Quando isto não ocorre, tem-se que a interação entre os programadores fica prejudicada. Sendo a comunicação o principal elemento para promover a percepção, quando barreiras são criadas para ela, então a percepção também é prejudicada. A percepção é um elemento relevante neste cenário pelo fato de contextualizar os programadores para a execução de suas atividades (Dourish e Bellotti, 1992). Ela permite identificar quem está trabalhando no projeto, o que eles estão fazendo, em quais artefatos estão ou estavam manipulando, e como seus trabalhos poderão impactar outros trabalhos (Storey et al., 2006). Neste contexto, tem-se que a ausência de suporte a elementos de percepção no desenvolvimento distribuído de software pode prejudicar a execução de atividades de compreensão (Conceição, 2012).

O Collaborative SourceMiner (Conceição, 2012) é o resultado da combinação de um ambiente interativo baseado em múltiplas visões com elementos de percepção para apoiar a compreensão de software no desenvolvimento distribuído. O objetivo é

fornecer informações que permitam aos integrantes das equipes conhecerem, através do ambiente, o que os demais pesquisaram, manipularam ou alteraram em um determinado projeto de software.

Pesquisas prévias desenvolveram soluções para apoiar a percepção em uma infraestrutura distribuída de desenvolvimento e manutenção de sistemas. Por exemplo, Sarma et al. (2003) desenvolveram o conceito de ponderação. Ponderação é o que deve ser levado em consideração para promover a percepção nas interações dos programadores. Cada interação tem seu nível de impacto no estado do projeto compartilhado, e o que é ponderado é compartilhado com o objetivo de promover a percepção. O resultado desse trabalho é uma ferramenta chamada Palantír. O ProjectWatcher é o resultado de uma pesquisa elaborada por Schneider et al. (2004), cujo objetivo é disponibilizar históricos de interações. Além disso, provê diferentes visualizações de quem está ativo no projeto, em quais artefatos e atividades eles estão atuando. O Lighthouse (Silva et al., 2006) cria visualizações com informações coletadas dos espaços de trabalho dos programadores. Neste trabalho foi aplicado o conceito de *Emerging Design*, que é uma representação atualizada do projeto tal como existe no código dos programadores.

Apesar das soluções propostas pelas pesquisas em desenvolvimento distribuído de software, pouco tem sido explorado no que diz respeito à utilização de ambientes interativos baseados em múltiplas visões. Para tanto, a solução proposta pelo Collaborative SourceMiner considera que programadores geograficamente dispersos necessitam de um conjunto diversificado de visões para apoiar a atividade de compreensão.

Este artigo está organizado da seguinte forma: a seção 2 apresenta os elementos relevantes para a contextualização deste artigo: desenvolvimento distribuído de software (DDS), percepção e compreensão de software. A seção 3 apresenta o modelo conceitual proposto. A seção 4 apresenta um estudo de caso para analisar como programadores identificam anomalias de modularidade de software em um cenário distribuído. Por fim, a seção 5 apresenta as considerações finais.

2. Compreensão e Percepção em Desenvolvimento Distribuído de Software

O desenvolvimento distribuído de software tem como um dos principais objetivos o ganho de produtividade, a redução de custos e a melhoria na qualidade do software (Prikladnicki e Audy, 2007). Entretanto, dentre as dificuldades inerentes ao desenvolvimento de software, tem-se que a forma como se dá a comunicação entre os participantes como uma questão importante a ser analisada. Sendo a percepção dependente da comunicação, tem-se que a percepção pode ser dificultada em ambientes de desenvolvimento distribuído.

A compreensão de software consiste na obtenção de conhecimento das funcionalidades, estrutura e comportamento de um sistema de software (Mayrhauser e Vans, 1993). No desenvolvimento distribuído, as atividades de compreensão podem ser dificultadas, por exemplo, pelos seguintes motivos: determinar quem possui conhecimento sobre diferentes partes do projeto (Zimmermann e Selvin, 1997), e comunicar com outros integrantes da equipe que atuam em horários diferentes (Herbsleb et al., 1999).

Elementos de visualização de software têm sido propostos na literatura para apoiar

a compreensão de software (Pacione, 2004; Lintern et al., 2003). Entretanto, o que se tem verificado é a necessidade de suporte nestas propostas à colaboração.

Conforme evidenciado por Gutwin e Greenberg (2002), o espaço de trabalho tem um papel importante durante a colaboração, mantendo o conhecimento sobre as interações dos outros, resultando em informações para a percepção. Omoronyia (2009) propõe que os benefícios da divisão das atividades de desenvolvimento distribuído podem ser encontrados capturando as trilhas de interações que ocorrem dentro dos espaços de trabalho. Essas trilhas podem ser construídas quando programadores atuam sobre suas tarefas de desenvolvimento diário deixando para trás vestígios de histórico daquilo que realizaram.

Em ambientes de programação distribuída, a utilização de sistemas colaborativos pode ser útil para apoiar a coleta e disponibilização de informações a respeito das interações no espaço de trabalho. Tais informações podem ser integradas com o conhecimento existente para manter um senso de percepção. Este senso permite, por exemplo, identificar em qual artefato o outro integrante está atuando e o que ele está fazendo. Quando este conhecimento se associa àqueles adquiridos com a interpretação de metáforas visuais, têm-se como resultado a geração de significados compartilhados sobre os artefatos para potencializar atividades de compreensão de software (Conceição, 2012).

3. O Modelo Conceitual Proposto

A Figura 1 ilustra o modelo conceitual para a solução proposta neste trabalho. A **parte A** da figura indica a atuação dos programadores na compreensão do código fonte utilizando o ambiente de desenvolvimento de software (ADS) Eclipse. O ADS é utilizado em conjunto com o plug-in Collaborative SourceMiner, representado na figura pelo círculo vermelho. A **parte B** ilustra o uso combinado das visões enriquecidas com dados obtidos dos elementos de percepção. Estes dados são provenientes das "mensagens de usuário" e "mensagens de sistema". A **parte C** indica que os dados da aplicação analisada obtidos do modelo Java (*Java Model*) poderão ser enriquecidos por métricas relacionadas ao tamanho, complexidade ciclomática, acoplamento, entre outras. A **parte D** ilustra que a interação com os cenários visuais é viabilizada através dos filtros de controle. O resultado esperado é um conjunto de visões enriquecidas com elementos de percepção para apoiar as atividades de compreensão de software, conforme ilustrado na **parte E**.

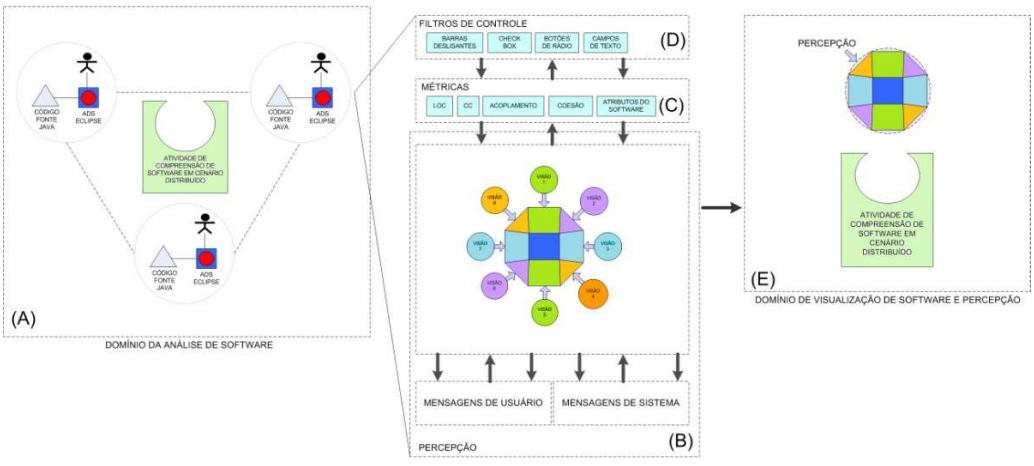


Figura 1. O Modelo Conceitual para o Collaborative SourceMiner

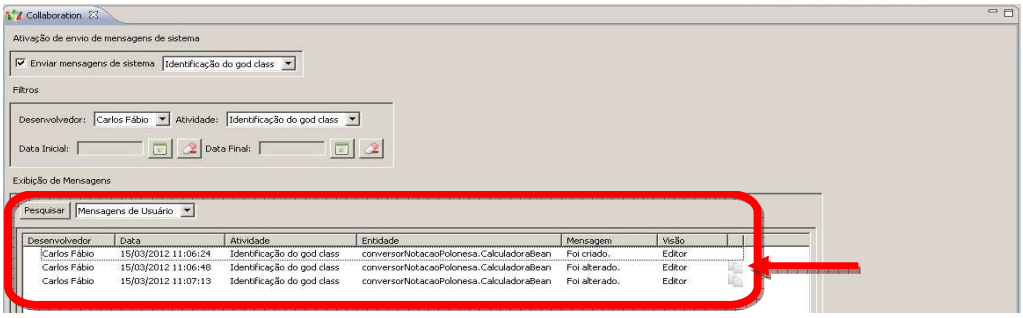


Figura 2. Visão Collaboration Detalhando as Ações de um Programador

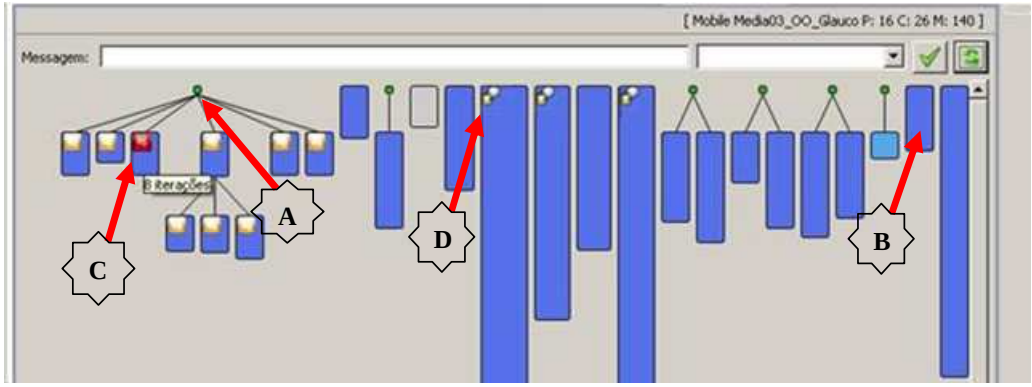


Figura 3. Visão Polimétrica Exibindo Mensagens de Usuário e Sistema

O modelo possibilita a comunicação semi-síncrona. As interações ocorrem tanto através de mensagens criadas por programadores como também de mensagens geradas pelo sistema. O objetivo das mensagens de usuário é registrar informações julgadas relevantes para contextualizar os programadores. As mensagens de sistema são sequências de eventos coletadas a partir de ações executadas pelos programadores. A partir do momento que um programador indica o início da execução de uma atividade, suas ações serão coletadas e enviadas para um servidor.

No quadro em destaque na Figura 2 são apresentados exemplos de mensagens de sistema. Cada registro exibe as seguintes informações: programador, data e horário, atividade, entidade (classe, interface, ou método) e visão utilizada. Estas informações servem para caracterizar as ações executadas por um determinado programador. Quando a ação executada representa uma modificação na composição de uma entidade, o ícone apontado pela seta é exibido. Este ícone pode ser selecionado para que sejam exibidas as versões anterior e posterior à alteração. Estas informações têm o objetivo de contextualizar os programadores que atuam em uma mesma atividade.

A Figura 3 exibe a visão polimétrica (Carneiro, 2011) que representa a hierarquia de herança das entidades (classes e interfaces) de um sistema. As interfaces são representadas pelos círculos esverdeados (seta A da figura). As classes são representadas por retângulos azuis (seta B da figura). Este é um exemplo de como as visões podem ser enriquecidas com elementos de percepção. Para esta finalidade, os ícones indicados pelas setas C e D são os recursos visuais utilizados para indicar a existência de mensagens associadas à entidade. Neste caso, a seta C indica o ícone específico para representar mensagens de sistema, enquanto que a seta D indica o ícone

que representa mensagens de usuário. De acordo com a figura, os ícones que representam as mensagens de sistema podem variar na tonalidade da cor vermelha. Ícones mais escuros destacam as entidades que um determinado programador mais interagiu. Desta forma, quem visualiza este cenário tem uma indicação inicial das entidades que a princípio estão mais relacionadas a uma atividade.

4. Um Estudo de Caso com o Collaborative SourceMiner

Um estudo de caso foi realizado com o objetivo de analisar a seguinte pergunta de pesquisa: "Como os elementos de apoio à percepção presentes no Collaborative SourceMiner auxiliam a compreensão de software considerando que os programadores atuam de forma colaborativa em locais geograficamente distribuídos?". O estudo contou com seis participantes, divididos em duas equipes com três em cada, não havendo interação entre as equipes. Cada equipe foi solicitada a identificar as anomalias de modularidade de software *Feature Envy*[1] (Fowler, 1999), *God Class*[2] (Riel, 1996) e *Divergent Change*[3] (Fowler, 1999). A aplicação selecionada foi o Mobile Media (MobileMedia, 2006). Todos os participantes selecionados para o estudo tinham conhecimento da linguagem Java e já tinham utilizado o ADS Eclipse. Durante a execução das atividades solicitadas, eles responderam um questionário para a coleta de dados. Estes dados foram utilizados nas avaliações quantitativa e qualitativa.

4.1. Análise e Discussão dos Resultados

A Tabela 1 apresenta os valores de precisão (p) e cobertura (c) de cada participante das equipes na identificação das anomalias de modularidade de software. A precisão quantifica a taxa de anomalias de modularidade corretamente identificadas pelo número de anomalias de modularidade detectadas. A cobertura quantifica a taxa de anomalias de modularidade corretamente identificadas pelo número de anomalias de modularidade da lista de referência já adotada em outros estudos (Carneiro, 2011). PA1, PA2 e PA3 representam os participantes 1, 2 e 3 da primeira equipe. PA4 e PA6 representam os participantes 1 e 3 da segunda equipe. PA5 não respondeu o questionário da forma solicitada, por este motivo seus valores não são apresentados neste estudo.

Tabela 1: Resultados por Participante das Equipes

	PA1		PA2		PA3		PA4		PA6	
	c	p	c	p	c	p	c	p	c	p
GC	0,9	1	0,8	0,9	0,9	0,7	0,2	0,7	0,2	0,7
DC	0,2	0,6	0,1	0,4	0,1	0,1	0,1	0,1	0,1	0,4
FE	0,4	0,3	0,1	0,1	0,2	0,2	0,2	0,4	0,2	0,6

A análise dos dados apresentados na Tabela 1 permite concluir que a maior variação

[1] *Feature Envy* ocorre quando um trecho de código de uma classe parece fazer mais parte de outra classe do que aquela na qual está contido (Fowler, 1999).
[2] *God Class* é caracterizado pela não coesão de comportamento e forte tendência de uma classe em atrair mais funcionalidades (Riel, 1996).
[3] *Divergent Change* ocorre quando uma classe necessita ser alterada frequentemente por diferentes razões (Fowler, 1999).

de precisão e cobertura ocorreu na equipe 1. Neste caso, o valor da precisão obtido por PA1 para *Divergente Change* foi de 0,6 e de 0,1 para PA3. Já na segunda equipe ocorreram duas variações nos valores de precisão e cobertura. A primeira relacionada à anomalia *Divergent Change* de 0,1 (PA4) para 0,4 (PA6). A segunda relacionada à anomalia *Feature Envy* de 0,4 (PA4) para 0,6 (PA6). Esta variação pouco acentuada pode ser atribuída à colaboração que ocorreu entre os participantes. Outra observação relevante foi a de que a anomalia *God Class* com os maiores valores de precisão e cobertura foi a que apresentou reduzida variação se considerados os valores de cada participante de cada equipe. A Tabela 2 apresenta as indicações dos participantes da primeira equipe para a identificação de *God Class*. Estão em destaque, sublinhadas e em negrito, as indicações convergentes dos participantes. Através dos relatos dos participantes, das evidências da comunicação que ocorreu através do ambiente, e da sequência de ações registradas foi possível identificar que os elementos de percepção disponíveis na ferramenta foram fundamentais para o alto nível de convergência.

Tabela 2: Indicações dos Participantes da Eq.1 de Ocorrências de God Class

Participante 1				
Versão 3	**Versão 4**	**Versão 5**	**Versão 6**	**Versão 7**
BaseController	**BaseController**	**ImageAcessor**	PhotoController	**MediaController**
ImageAcessor	**ImageAcessor**		**ImageAcessor**	
Participante 2				
BaseController	**BaseController**	**ImageAccessor**	**ImageAccessor**	**MediaController**
ImageAccessor	**ImageUtil**	AlbumData		
Participante 3				
BaseController	**BaseController**	**ImageAcessor**	**ImageAcessor**	**MediaAcessor**
InvalidImageDataException	InvalidImageDataException	AlbumData		**MediaController**
ImageAcessor	**ImageAcessor**			
PhotoController				

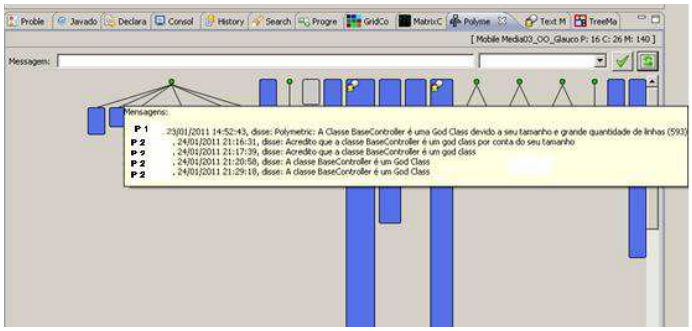


Figura 4. Comunicação entre os Participantes Utilizando a Visão Polimétrica

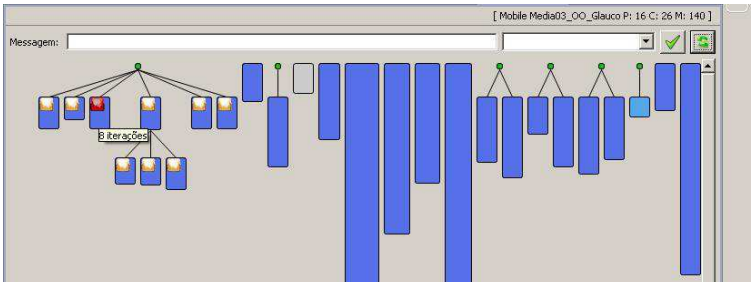


Figura 5. Visão Polimétrica Exibindo Mensagens de Sistema

Os trechos de conversa entre os participantes apresentados na Figura 4 revelam evidências da comunicação durante a execução das atividades. A Figura 5 exibe a visão polimétrica vista por PA2 na identificação de *Feature Envy*. Para obter esta visão PA2

executou uma consulta utilizando as seguintes opções de filtro: participante-PA1 e atividade-*Feature Envy*. Tendo como base esta visão, PA2 relatou: “após perceber que PA1 interagiu mais vezes com a classe *UnavailablePhotoAlbumException* acessei-a para verificar se de fato se tratava de uma ocorrência de *Feature Envy*.” O resultado da análise foi que a classe em questão não era uma ocorrência de *Feature Envy*.

Após a conclusão das atividades, PA1 relatou: “Os comentários dos outros participantes me auxiliaram a perceber algumas características de classes antes não observadas por mim.”. PA2 relatou: “Quando os outros participantes já haviam identificado as classes, eu procurava outras classes, ou conferia as classes identificadas. Era como resolver um exercício em grupo. Fiquei com algumas dúvidas, mas aprendi através do acesso às ações e respostas registradas por meus colegas”. Já PA3 relatou: “As mensagens incisivas, principalmente as de PA1, foram fundamentais para tirar possíveis dúvidas que eu tive na execução da atividade”. PA4 também relatou: “A indicação da classe *BaseController* foi em concordância com a resposta de PA6.”. Estes comentários indicam evidências iniciais de que o enriquecimento das representações visuais com as dicas dos participantes contribuiu para a geração de um entendimento compartilhado.

As seguintes observações foram obtidas considerando a pergunta de pesquisa deste estudo e a análise dos resultados. **Observação 1**: durante a execução das atividades, os participantes se comunicaram e, na maioria das vezes, convergiram em suas indicações. **Observação 2**: os resultados apresentam evidências iniciais de que os participantes inicialmente buscavam no ambiente a existência de indícios que os fizessem conhecer a respeito das indicações dos outros. Quando encontravam, faziam uma análise dessas indicações para, então, elaborar as suas. **Observação 3**: os resultados apresentam evidências iniciais de que, devido aos elementos de percepção presentes na ferramenta, os participantes adotaram estratégias parecidas na identificação das anomalias. Ou todos adotaram uma estratégia otimista, ou então adotaram uma estratégia pessimista.

Ameaças à Validade do Estudo. O tamanho e a complexidade do objeto de estudo (o Mobile Media) estão aquém daquelas relacionadas às aplicações comerciais típicas. Entretanto, esta aplicação já tem sido utilizada em outros estudos a exemplo de Carneiro (2011) para fins da caracterização do uso de ambientes de visualização de software. O número reduzido de participantes pode comprometer a representatividade dos resultados obtidos e a sua generalização. O número de participantes considerou o equilíbrio entre o custo do estudo e a análise qualitativa dos resultados para derivar as observações.

5. Conclusão

Este artigo apresentou um estudo de caso para caracterizar o uso de recursos de visualização enriquecidos com elementos de percepção para a compreensão de software em um ambiente de desenvolvimento distribuído. O estudo foi conduzido com duas equipes de três participantes cada que atuaram em um ambiente de desenvolvimento distribuído. Para a execução das atividades os participantes utilizaram exclusivamente o Collaborative SourceMiner como suporte à percepção. Os resultados do estudo apresentaram evidências iniciais da efetividade do uso de elementos de percepção para a compreensão de software de forma colaborativa. O modelo conceitual e a implementação da solução proposta estão sendo revistos tendo como referência as oportunidades de melhorias identificadas no estudo descrito neste artigo. A próxima etapa será o planejamento de um novo estudo com foco no suporte à coordenação em

ambientes de desenvolvimento distribuído.

Referências

Carneiro, G. F. (2011) “SourceMiner: Um Ambiente Integrado Para Visualização Multi-Perspectiva De Software”. 229 f. Tese de Doutorado em Ciência da Computação. Universidade Federal da Bahia.

Conceição, C.F.R. (2012) “Analisando o Uso de Elementos de Percepção Para Atividades de Compreensão de Software em um Ambiente de Desenvolvimento Distribuído”. Dissertação de Mestrado, UNIFACS, Salvador.

Dourish, P.; Bellotti, V. (1992) “Awareness and coordination in shared workspace”, Conference on Computer-Supported Cooperative Work. pp. 107-114, Toronto, Canada, Nov.

Fowler, M. (1999) “Refactoring: Improving the Design of Existing Code”. Addison-Wesley Professional.

Herbsleb, J.; Grinter, R.; Perry, D. (1999) “The geography of coordination: dealing with distance in R&D work”. Proc. ACM SIGGROUP conference on supporting group work.

Gutwin, C.; Greenberg, S. (2002) “A Descriptive Framework of Workspace Awareness for Real-Time Groupware”. Journal of Computer-Supported Cooperative Work. Issue 3-4. p. 411-446.

Lintern R.; Michaud J.; Storey, M.; Wu, X. (2003) “Plugging-in Visualization: experiences integrating a visualization tool with Eclipse”. Proceedings of the 2003 ACM symposium on Software visualization, pp. 47-56, New York.

Mayrhauser, A. V.; Vans, A. M. (1993) “From Code Understanding Needs to Reverse Engineering Tool Capabilities”. Proceedings of the 6th International Workshop on Computer-Aided Software Engineering, pp. 230-239.

MOBILEMEDIA. MobileMedia. Disponível em http://mobilemedia.sourceforge.net. 2006.

Omoronyia, I. (2009) “Using Developer Activity Data to Enhance Awareness during Collaborative Software Development”. University of Strathclyde. Glasgow, p. 50.

Pacione, M. J. (2004) “Software visualisation for object-oriented program comprehension”. In Doctoral Symposium, Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pp. 63-65, Edinburgh. IEEE / ACM.

Prikladnicki, R.; Audy, J. L. N. (2007) “Um Modelo de Referência para Desenvolvimento Distribuído de Software”. 16 f. Artigo (3º) - Curso de Informática, Departamento de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre.

Riel, A. J. (1996) “Object-oriented design heuristics”. Addison-wesley professional.

Sarma, A.; Noroozi, Z.; Hoek, A. (2003) “Palantír: raising awareness among configuration management workspaces”. In International Conference on Software Engineering (ICSE ‘03). IEEE Computer Society. pp. 444-454.

Silva, I. et al. (2006) “Lighthouse: coordination through emerging design”. In ETX 2006 (OOPSLA Workshop On Eclipse Technology Exchange), pp. 11 – 15.

Storey, M.-A; Cheng, L.-T; Rigby, P.; Bull, R.I. (2006) “Shared Waypoints and Social Tagging to Support Collaboration in Software Development”, In Proceedings of Computer Supported Cooperative Work. Banff, Canada, 195-198.

Zimmermann B.; Selvin, A. M. (1997) “A framework for assessing group memory approaches for software design projects”. Proc. Conference on Designing interactive systems.