REALIZAÇÃO: SBC Sociedade Brasileira de Computação

EXECUÇÃO: UEM Universidade Estadual de Maringá; UTFPR Universidade Tecnológica Federal do Paraná Câmpus Campo Mourão; DIN Departamento de Informática

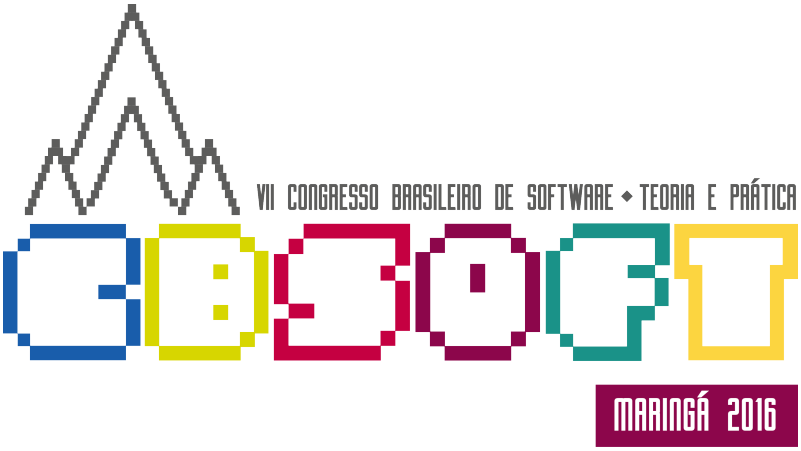
ORGANIZAÇÃO: TASA EVENTOS

APOIO: Convention & Visitors Bureau Maringá e Região

FOMENTO: CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico; Paraná Governo do Estado Secretaria da Ciência, Tecnologia e Ensino Superior; Fundação Araucária Apoio ao Desenvolvimento Científico e Tecnológico do Paraná; UEM Universidade Estadual de Maringá; CAPES

PATROCINADORES GIGA: UniCesumar; COLIVRE Cooperativa de Tecnologias Livres

PATROCINADORES MEGA: ESPWEB Especialização em Web e Mobile; ThoughtWorks



VII CONGRESSO BRASILEIRO DE SOFTWARE: TEORIA E PRÁTICA (CBSOFT 2016) – SESSÃO DE FERRAMENTAS

19 a 23 de setembro de 2016 | *September 19 - 23, 2016*
Maringá, PR, *Brazil*

ANAIS | *PROCEEDINGS*

Sociedade Brasileira de Computação – SBC

COORDENADOR DO COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE CHAIR*
Fabiano Cutigi Ferrari (UFSCar)

EDITORES | *PROCEEDINGS CHAIRS*
Marco Aurélio Graciotto Silva (UTFPR)
Willian Nalepa Oizumi (IFPR)

COORDENADORES GERAIS | *GENERAL CHAIRS*
Edson Oliveira Júnior (UEM)
Thelma Elita Colanzi (UEM)
Igor Steinmacher (UTFPR)
Ana Paula Chaves Steinmacher (UTFPR)
Igor Scaliante Wiese (UTFPR)

REALIZAÇÃO | *REALIZATION*
Sociedade Brasileira de Computação (SBC)

EXECUÇÃO | *EXECUTION*
Universidade Estadual de Maringá (UEM) – Departamento de Informática (DIN)
Universidade Tecnológica Federal do Paraná (UTFPR) – Câmpus Campo Mourão (UTFPR-CM)

ISBN: 978-85-7669-340-6

Apresentação

A Sessão de Ferramentas do CBSoft proporciona um fórum para a apresentação e a demonstração de soluções automatizadas que apoiam o processo de desenvolvimento e gerenciamento de software em suas mais diversas necessidades e manifestações. O público-alvo inclui membros da comunidade acadêmica e da indústria. Os acadêmicos apresentam os resultados de seus projetos de pesquisa aplicados nas seguintes áreas: Engenharia de Software; Linguagens de Programação; Componentes, Arquiteturas e Reutilização de Software; e Teste de Software. Profissionais da indústria apresentam ferramentas comerciais ou de uso interno que trazem ganhos significativos de produtividade e/ou qualidade para atividades do processos de desenvolvimento de software.

Em 2016, o Comitê de Programa (TPC) foi formado por membros de edições anteriores, os quais também indicaram novos membros ativos da comunidade do CBSoft. Todos participaram ativamente da avaliação dos 32 trabalhos submetidos. Desse total, 28 trabalhos tiveram 4 revisões e apenas 4 tiveram 3 revisões. Dos 17 trabalhos selecionados para apresentação, apenas 2 tiveram uma avaliação como "weak reject", porém ambos tiveram "strong accept" de pelo menos um revisor, o que os levou para uma boa colocação no ranking final. Os demais 15 trabalhos aceitos tiveram todas as avaliações positivas, variando entre "strong accept" e "weak accept". Em relação à língua adotada para a redação dos trabalhos, 11 em inglês (7 entre os aceitos) e 21 em português (10 entre os aceitos).

As apresentações foram divididas em 4 sessões técnicas, abordando os seguintes temas: Manutenção, Reúso, Gestão de Configuração e Equipes, Requisitos, Linguagens, Compiladores, Verificação & Validação, MDD e Métricas.

Por fim, ressalta-se que a realização da Sessão de Ferramentas em 2016 só foi possível pelo envolvimento, engajamento e profissionalismo da comunidade, incluindo autores, revisores e organizadores do CBSoft 2016. A coordenação da Sessão de Ferramentas registra, em nome da comunidade, os sinceros agradecimentos a todos.

Maringá, setembro de 2016.

Fabiano Cutigi Ferrari (UFSCar)
Coordenador da Sessão de Ferramentas do CBSoft 2016

Foreword

The CBSoft Tools Session provides a forum for the presentation and demonstration of automated solutions to support the software development and management processes in a variety of needs and contexts. The audience includes members from both academia and industry. Academics present results from their research applied to Software Engineering; Programming Languages, Components, Architectures and Software Reuse; and Software Testing. Professionals from industry present proprietary tools that bring substantial gains in productivity and/or quality with respect to the software development process.

In 2016, the Program Committee included members from previous editions and new, active members from the CBSoft community. All TPC members participated actively from the evaluation of the 32 submitted papers. In total, 28 papers were evaluated with 4 reviews and only 4 papers with 3 reviews. From the 17 accepted papers, only 2 were evaluated with a "weak reject"; however, both of them received a "strong accept" from at least one of the remaining Reviewers, thus lifting up these 2 papers to a good position in the final ranking. The other 15 accepted papers received all positive evaluations (i.e. "strong accept" or "weak accept"). From the 32 submitted papers, 11 were written in English (7 accepted) and 21 were written in Portuguese (10 accepted).

Presentations were grouped in 4 technical sessions, addressing the following topics: Maintenance, Reuse, Team and Configuration Management, Requirements, Languages, Compilers, Verification and Validation, MDD and Metrics.

Last but not least, the program of the Tools Session 2016 was feasible particularly due to the commitment and professionalism of the whole community, including authors, reviewers and CBSoft organizers. The Chair of the Industry Track thank them all on behalf of the community.

Maringá, September 2016.

Fabiano Cutigi Ferrari (UFSCar)
Chair – CBSoft 2016 Tools Session

Comitê técnico | *Technical committee*

Coordenador de comitê de programa | *PC chair*

Fabiano Cutigi Ferrari (UFSCar)

Comitê de programa | *Program committee*

Adenilso Simão (ICMC/USP)
Alberto Costa Neto (UFS)
Alexandre Mota (UFPE)
Americo Sampaio (UNIFOR)
Anamaria Martins Moreira (UFRJ)
Arilo Dias Neto (UFAM)
Auri Vincenzi (UFSCar)
Bruno Costa (UFRJ)
Cecília Rubira (UNICAMP)
Cláudio Sant'Anna (UFBA)
Daniel Lucrédio (UFSCar)
David Déharbe (UFRN)
Delano Beder (UFSCar)
Eduardo Figueiredo (UFMG)
Elder José Cirilo (UFSJ)
Elisa Huzita (UEM)
Fernando Figueira Filho (UFRN)
Fernando Trinta (UFC)
Franklin Ramalho (UFCG)
Glauco Carneiro (UNIFACS)
Gledson Elias (UFPB)
Ingrid Nunes (UFRGS)
Juliana Saraiva (UFPB)
Lincoln Rocha (UFC)
Luis Ferreira Pires (University of Twente)
Marco Tulio Valente (UFMG)
Maria Istela Cagnin (UFMS)
Martin Musicante (UFRN)
Márcio Cornélio (UFPE)
Patricia Machado (UFCG)
Paulo Maciel (UFPE)
Paulo Maia (UECE)
Paulo Pires (UFRJ)
Pedro Santos Neto (UFPI)
Ricardo Lima (UFPE)

Rita Suzana Pitangueira Maciel (UFBA)
Rohit Gheyi (UFCG)
Rosângela Penteado (UFScar)
Sandra Fabbri (UFSCar)
Tiago Massoni (UFCG)
Uirá Kulesza (UFRN)
Valter Camargo (UFSCar)
Vander Alves (UnB)
Vania Neris (UFSCar)

Revisores externos | *External reviewers*

Adriano dos Santos
Ana Patricia Fontes Magalhaes Mascarenhas
Anderson Belgamo
André Hora
Augusto Zamboni
Bruno Santos
Bruno Silva
Carlos Damasceno
Daniel San Martin
Erica Sousa
Erik Antonio
Ernesto Matos
Gustavo de Souza Santos
Heitor Costa
Johnatan Oliveira
Kattiana Constantino
Kenyo Faria
Marcelo Alves
Paulo Parreira Júnior
Rafael Oliveira
Rubens de Souza Matos Júnior
Wagner Vieira

Artigos técnicos | *Technical papers*

Sessão 1: Manutenção, Reúso e Gestão de Configuração e Equipes

Sessão 2: Requisitos, Linguagens e Compiladores

Sessão 3: V&V-1 e MDD

Sessão 4: V&V-2 e Métricas

DCEVizz: Uma Ferramenta para Visualização de Código Morto na Evolução de Sistemas de Software Java

Camila Bastos, Paulo Afonso Júnior, Heitor Costa

Departamento de Ciência da Computação - Universidade Federal de Lavras - MG - Brasil

camilabastos@posgrad.ufla.br, pauloa.junior@dcc.ufla.br, heitor@dcc.ufla.br

Resumo. *A evolução é essencial para sistemas de software não se tornarem insatisfatórios e está relacionada com o aumento da sua complexidade ao longo das versões. Para reduzir a complexidade e facilitar a compreensão, técnicas de visualização de software podem ser utilizadas para representar informações relacionadas a esses sistemas. Um dos fatores que contribui com essa complexidade é a presença de código morto. A identificação, a visualização e a compreensão desse código na evolução do software podem auxiliar a reduzir a complexidade e predizer o aumento desse problema em versões futuras, possibilitando a tomada de medidas preventivas. Com isso, a relação negativa entre evolução e aumento da complexidade do software é reduzida e os custos de manutenção são diminuídos. Considerando esses fatores, neste artigo, é apresentado DCEVizz, um plug-in que detecta código morto em versões de sistemas de software Java e apresenta os resultados utilizando técnicas de visualização de software, destacando suas características evolutivas.*

Abstract. *The evolution is necessary for software to be satisfactory and is related with increasing complexity of versions. To reduce complexity and ease of comprehension, software visualization techniques are used to represent information related to software. One of the factors that contributes to this complexity is the presence of dead code. The identification, visualization and understanding of dead code in the evolution of software can be useful to reduce complexity and to predict the increase of this problem in future versions, allowing the execution of preventive actions. Thus, the negative relationship between evolution and increasing software complexity is reduced and maintenance costs are reduced. Considering these factors, this paper presents DCEVizz, a plug-in that detects dead code in Java software versions and displays the results using software visualization techniques, highlighting its evolutionary characteristics.*

Link do Vídeo: *https://www.youtube.com/watch?v=Ax7Gyi6ebPw&feature=youtu.be*

1. Introdução

A evolução é essencial para sistemas de software não se tornarem insatisfatórios e refletirem as constantes mudanças que ocorrem no ambiente no qual ele está inserido [Lehman; Belady, 1985]. Informações históricas são geradas ao longo dessa evolução e podem ser utilizadas para encontrar a origem de problemas atuais ou para prever características futuras desses sistemas [D'Ambros, 2008]. Abordagens baseadas em

informações evolutivas mostraram que existe relação entre a evolução de sistemas de software e o aumento da complexidade e da poluição do código em suas versões [Gold; Mohan, 2003; Burd; Rank, 2001]. Com isso, técnicas de visualização de software podem ser utilizadas para representar informações relacionadas à evolução para reduzir complexidade e facilitar compreensão [Novais *et al*., 2013; Rufiange; Melancon, 2014].

Um dos fatores que contribui com o aumento dessa poluição é a presença de código morto, que pode ser considerado como trechos de código que nunca são executados. Existem diferentes tipos de código morto, no qual a granularidade de métodos pode ser considerada a mais prejudicial para a manutenibilidade de sistemas de software. Isso ocorre por possuírem maior quantidade de linhas de código, não estarem vinculados a requisitos e contribuírem com a existência de código não testado [Martin, 2008]. Métodos mortos prejudicam a compreensão do software, colaborando para a manutenção ser considerada a etapa mais cara do ciclo de vida, pois metade do tempo utilizado é destinado à compreensão [Ducasse; Lanza, 2005; Scanniello, 2014]. Nesse contexto, a visualização e a análise da evolução do código morto podem ser utilizadas para entender o surgimento desse problema nas versões de sistemas de software e prever o aumento da poluição das versões futuras.

Algumas ferramentas apresentam a evolução de características do software utilizando técnicas de visualização e outras ferramentas automatizam a detecção de código morto. No entanto, não foram encontradas ferramentas que detectam e analisam a evolução do código morto. Considerando a importância da automatização dessa detecção e a utilidade dessa análise, neste artigo, é apresentado DCEVizz (*Dead Code Evolution Visualization*), um *plug-in* para o Eclipse que detecta estaticamente código morto (métodos) em versões de sistemas de software Java e os apresenta utilizando técnicas de visualização de software.

O restante do artigo está organizado da seguinte forma. Na Seção 2, é descrito o funcionamento do DCEVizz, uma visão geral da sua arquitetura e um exemplo de uso. Na Seção 3, são apresentadas ferramentas relacionadas, mostrando seu diferencial em relação ao DCEVizz. Na Seção 4, são apresentadas as considerações finais.

2. O *Plug-in* DCEVizz

DCEVizz é um *plug-in* para o Eclipse que detecta estaticamente código morto (métodos nunca executados) em versões de sistemas de software e permite observar a evolução desse código por meio de técnicas de visualização de software. Esse *plug-in* pode ser útil aos mantenedores de software na execução das seguintes atividades: i) identificar métodos mortos presentes em versões de software; ii) identificar versão, pacote ou classe com maior quantidade de métodos mortos, podendo priorizá-los na manutenção; iii) compreender a evolução do software, especificamente a degradação da qualidade interna por causa da presença de código morto; iv) ter mais segurança na eliminação do código morto, pois, se um método é morto em várias versões, reforça o indicativo de sua não utilização; v) possibilitar a tomada de decisões e medidas preventivas para evitar que versões futuras do software possuam código morto; e vi) eliminar código morto em versões em análise.

2.1 Funcionamento

DCEVizz é baseado no modelo de referência de visualização da informação e organizado em três etapas [Card *et al*., 1999] (Figura 1):

- Na etapa **Fonte de Dados**, o código das versões do software é mapeado para estruturas de dados do tipo árvore utilizando *plug-ins* do JDT (*Java Development Tools*). Nessa estrutura, os nós representam elementos do software, tais como, pacotes, classes e métodos e as ligações entre os nós correspondem aos relacionamentos entre esses elementos;

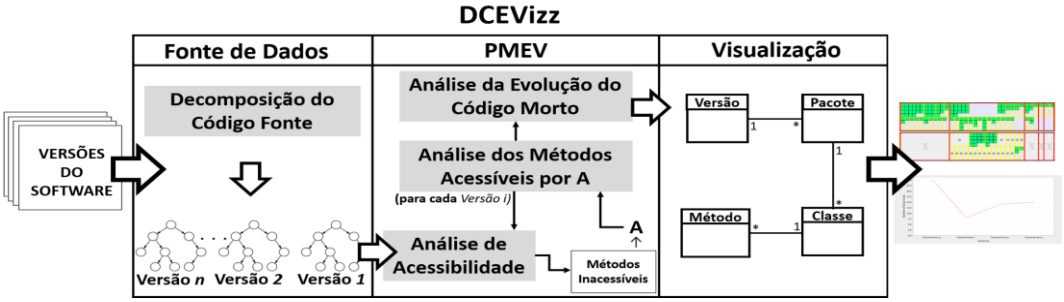


Figura 1. Funcionamento do DCEVizz

- Na etapa **Processamento e Mapeamento em Estruturas Visuais (PMEV)**, inicia-se a detecção estática de código morto para identificar métodos inacessíveis (mortos). A técnica indicada para essa detecção é Análise de Acessibilidade [Bastos *et al*., 2016]. Nessa técnica, uma hierarquia de chamadas é gerada com apoio do JDT para cada método presente na árvore. Essa hierarquia é representada por um grafo direcionado, cujos nós correspondem aos métodos e arestas correspondem às chamadas existentes entre esses métodos. Os nós sem arestas incidentes são considerados inacessíveis por não possuírem chamadas de outros métodos, sendo definidos como código morto. Após identificar o conjunto `A` de métodos inacessíveis, é realizada uma análise para identificar métodos acessíveis apenas por métodos pertencentes ao conjunto `A`, sendo também definidos como código morto. Essa análise é executada por meio de uma busca em largura no grafo da hierarquia de chamadas, considerando como vértice inicial cada método morto do conjunto `A`. As análises para identificar métodos inacessíveis são executadas em cada versão do software e as informações de localização de cada método (versão, pacote e classe) são armazenadas em estruturas de dados. Essas informações de localização são necessárias na análise da evolução do código morto, em que é verificada a ocorrência de cada método inacessível nas versões analisadas. Dessa forma, o mapeamento das características visuais de cada método morto é realizado e armazenado nas estruturas de dados;
- Na etapa **Visualizações**, as estruturas de dados criadas na etapa anterior são utilizadas na renderização das visualizações. Duas visualizações foram implementadas e integradas ao DCEVizz. Em uma, são apresentados os resultados sob o ponto de vista quantitativo, exibindo a quantidade de código morto existente em cada versão do software utilizando gráfico de linha gerado pela biblioteca JFreeChart (http://www.jfree.org/jfreechart/). Na outra, a visualização qualitativa foi elaborada realizando adaptações nas técnicas TreeMap [Turo; Johnson, 1992] e Matriz de Evolução [Lanza, 2001].

2.2 Arquitetura da Ferramenta

Na Figura 2, é apresentado o Diagrama de Pacotes que representa a arquitetura do DCEVizz. O pacote `Frames` contém as interfaces de usuário, as classes do pacote `Handler` possuem a responsabilidade de receber as requisições do usuário e de iniciar os componentes que analisam o código e geram as visualizações. As classes do pacote `CodeAnalysis` são responsáveis pela realização das atividades descritas nas etapas **Fonte de Dados** e **PMEV**. O pacote `Model` contém as classes de domínio que armazenam as informações necessárias para renderizar as visões. O pacote `View` contém as classes responsáveis pela renderização das visualizações do DCEVizz.

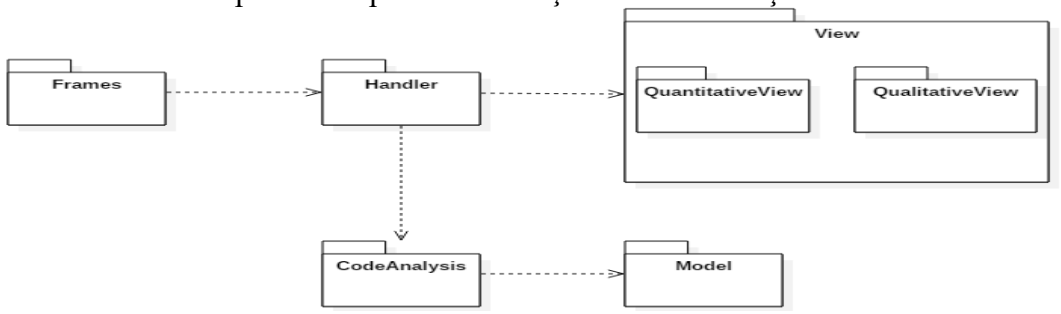


Figura 2. Diagrama de Pacotes do DCEVizz

2.3 Exemplo de Uso

Com o DCEVizz instalado no Eclipse, é disponibilizado um botão na barra de ferramentas, bem como uma opção na barra de *menu* que permite iniciar sua execução. As versões do software a serem analisadas devem estar no *workspace* e selecionadas utilizando a interface do *plug-in* (Figura 3).

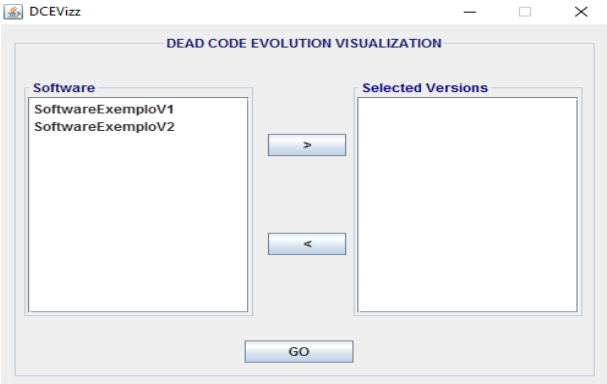


Figura 3. Tela Inicial do *Plug-in* DCEVizz

Para exemplificar a utilização do DCEVizz, foram utilizadas duas versões de um software real, identificado neste artigo como Software Exemplo (o nome do software foi omitido propositalmente). As versões 1 e 2 desse software contém, respectivamente, 7 e 8 pacotes, 27 e 42 classes e 346 e 532 métodos. Após a análise dessas versões, DCEVizz identificou 72 métodos mortos na primeira versão (tempo: 22 segundos) e 75 métodos mortos na segunda versão (tempo: 37 segundos); assim, o tempo médio foi aproximadamente 29 segundos (média aritmética). Mais especificamente, DCEVizz analisou a acessibilidade de 15,55 métodos e 14,37 métodos por segundo nas duas versões, respectivamente. Além disso, o processo de renderização das visualizações foi

instantâneo em ambas versões. O usuário do DCEVizz obtém essas informações por meio do arquivo de *log* gerado após a análise das versões selecionadas. Finalizada a detecção de código morto, duas *views* do Eclipse são carregadas contendo: i) **Visualização quantitativa** apresenta suscintamente a evolução da quantidade de código morto, facilitando a percepção do aumento/diminuição dessa quantidade ao longo das versões; e ii) **Visualização qualitativa** apresenta em detalhes a evolução do código morto, permitindo acompanhar suas mudanças ao longo das versões. Na visualização quantitativa do Software Exemplo (Figura 4), percebe-se "pequeno" aumento na quantidade de código morto entre as duas versões analisadas. Na visualização qualitativa (Figura 5), a evolução é mostrada pela organização das versões, dos pacotes, das classes e dos métodos em uma matriz, conforme sugerido pela técnica Matriz de Evolução.

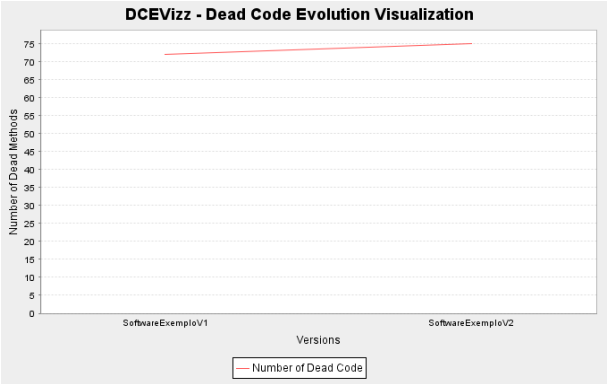


Figura 4. Representação Visual Quantitativa

Cada linha da matriz (bordas pretas) representa uma versão em análise e os pacotes (bordas vermelhas), classes (bordas amarelas) e métodos mortos (bordas pretas com espessura menor) são organizados nas colunas. A representação hierárquica dos métodos mortos visa facilitar sua localização no software e foi baseada na técnica TreeMap**.** Para facilitar a compreensão da evolução, os componentes do software foram organizados de maneira simétrica, ou seja, os mesmos pacotes, as mesmas classes e os mesmos métodos em cada versão são colocados na mesma posição vertical. Nessa visualização, as diferenças relacionadas ao código morto são destacadas visualmente utilizando cores e símbolos no preenchimento dos componentes. A numeração presente na Figura 5 destaca as possíveis representações:

- **Representação 1**. A cor cinza com o símbolo " X " significa que o pacote existiu em alguma das versões anteriores e foi removido na versão. No exemplo, o pacote existia na primeira versão e tinha métodos mortos, porém ele foi excluído na segunda versão;
- **Representação 2**. A cor cinza com o símbolo "⊗" significa que a classe existiu em alguma das versões anteriores e foi removida na versão. No exemplo, a classe existia na primeira versão e tinha métodos mortos, mas ela foi excluída na segunda versão;
- **Representação 3**. A cor verde significa que o método morto não existia na versão anterior. Essa cor é a única possível na primeira versão do software;
- **Representação 4**. A cor laranja significa que o método morto foi propagado de uma versão para outra. Ele existia na versão anterior e continua existindo na versão;
- **Representação 5**. A cor cinza significa que o método era morto na versão anterior e deixou de ser morto na versão;

- **Representação 6**. A cor cinza com o símbolo "⊗" significa que o método era morto na versão anterior e foi excluído na versão;
- **Representação 7**. O símbolo "⊙" significa que o método é morto por ser chamado apenas por métodos mortos.

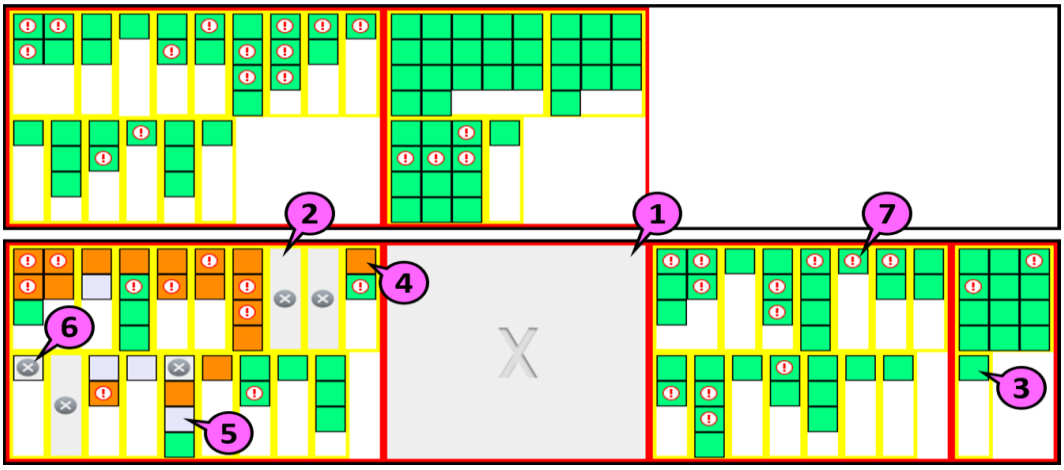


Figura 5. Representação Visual Qualitativa

Pacotes e classes que não possuem métodos mortos não são apresentados na visualização. Na Representação 1, o pacote foi representado pois, em pelo menos uma das versões anteriores, ele existia e tinha classes com métodos mortos. O mesmo acontece na classe apresentada na Representação 2, apesar dela não existir na segunda versão, ela foi representada na visualização. Na Representação 6, a classe não possui método morto e apenas foi representada visualmente pelo fato de que na versão anterior ela tinha métodos mortos. Além disso, pode-se perceber que o terceiro e quarto pacotes não existiam na primeira versão, sendo representados apenas na segunda versão com os métodos mortos destacados na cor verde. Caso a visualização exceda o tamanho da *view* da IDE Eclipse, barras de rolagens verticais e horizontais são criadas automaticamente.

Foram implementados alguns recursos de interação com a visualização qualitativa. Na Figura 6(A), é apresentada uma interação, na qual, ao clicar em cima de qualquer método morto, sua borda é destacada em todas as versões, permitindo ao usuário acompanhar sua evolução. Na Figura 6(B), há outra interação, na qual, ao clicar com o botão direito do *mouse* em cima de um método morto, aparece um *menu* com a opção de visualizar o código correspondente. Uma segunda opção aparece nesse *menu* se o método morto selecionado for acessível apenas por métodos mortos, permitindo o destaque na cor azul dos métodos chamadores (Figura 6(C)). Ao passar o *mouse* sobre pacotes, classes ou métodos, são apresentadas informações sobre esses componentes (Figura 6(D)). Além disso, são disponibilizados *menus* na *view* do Eclipse que permitem limpar os destaques da visualização, procurar métodos mortos e salvar o arquivo de *log* (Figura 6(E)).

3. Ferramentas Relacionadas

Algumas ferramentas detectam código morto em sistemas software Java. Outras permitem visualizar a evolução do software. No entanto, não foram encontradas ferramentas que permitem visualizar a evolução de código morto, conforme realizado

em DCEVizz. Por exemplo, JTombstone (http://jtombstone.sourceforge.net/) detecta métodos mortos e classes não utilizadas em apenas uma versão do software. Sua execução é via linha de comando e independe da IDE no qual o software foi implementado, fazendo necessário um arquivo de entrada XML (*eXtensible Markup Language*) com informações sobre quais componentes de software deverão ser analisados. UCDetector (http://www.ucdetector.org/) é um *plug-in* para o Eclipse que identifica métodos mortos, código em que a visibilidade possa ser alterada e atributos que possam ser definidos como constante. DUM-Tool é outro *plug-in* para o Eclipse desenvolvido para detectar métodos mortos em apenas uma versão do software [Romano; Scanniello, 2015]. Esse *plug-in* analisa *bytecodes* Java e identifica métodos mortos percorrendo uma representação do software baseada em grafo. O principal diferencial do DCEVizz em relação a essas ferramentas é a detecção de código morto em diferentes versões do software e análise da sua evolução.

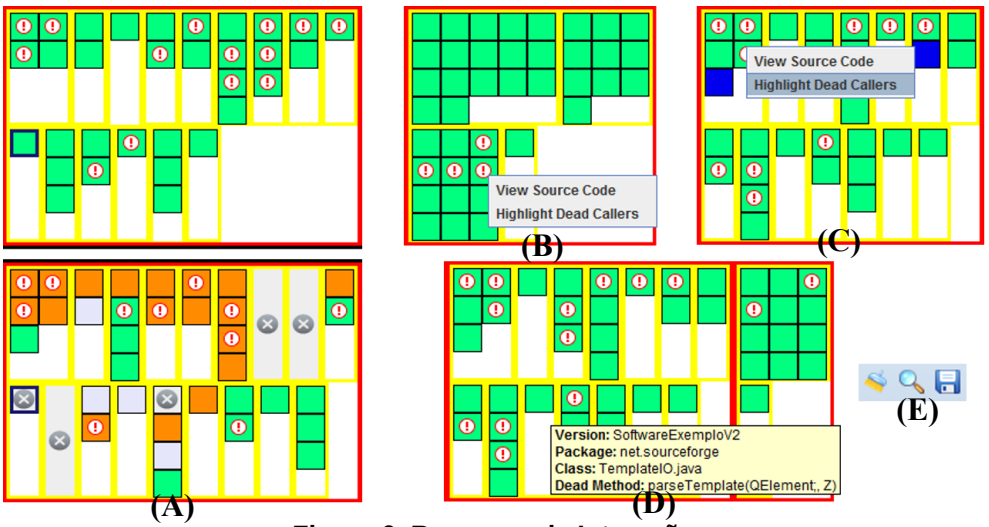


Figura 6. Recursos de Interação

SourceMiner Evolution é um *plug-in* para o Eclipse para visualizar a evolução [Novais *et al.*, 2013]. Esse *plug-in* permite acompanhar a evolução de *features* utilizando as técnicas TreeMap, Visão de Dependência, Visão Polimétrica e TimeLine Matrix. A compreensão da evolução ocorre com a utilização de cores que destacam o diferencial entre duas versões selecionadas pelo usuário. Ferramentas independentes de IDE também foram desenvolvidas para visualizar a evolução do software. Por exemplo, com a Devis [Zhi; Ruhe, 2013], pode-se acompanhar a evolução da documentação do software. Além disso, a Samoa [Minelli; Lanza, 2013] foi desenvolvida para visualizar a evolução de medidas em aplicações para dispositivos móveis. Ao contrário dessas ferramentas, DCEVizz visa apresentar especificamente a evolução de código morto utilizando técnicas de visualização de software.

4. Considerações Finais

Com o objetivo de auxiliar a compreensão da evolução do software em relação a código morto e na sua eliminação, neste artigo, foi apresentado DCEVizz, um *plug-in* para o Eclipse que analisa versões de sistemas de software Java para detectar código morto. Os resultados dessa análise são apresentados de forma quantitativa e qualitativa por meio de técnicas de visualização de software. Como limitação, DCEVizz identifica código morto

estaticamente e, por esse motivo, métodos acessíveis via reflexão e polimorfismo podem ser identificados indevidamente como inacessíveis.

Como trabalhos futuros, espera-se implementar recursos de filtragem para o usuário selecionar os pacotes que deseja analisar/visualizar. Espera-se analisar versões de sistemas de software Java coletados de vários repositórios de código aberto utilizando DCEVizz e outra ferramenta de detecção de métodos mortos para avaliar tempo e cobertura de detecção de código morto. Além disso, espera-se executar avaliações com usuários finais para investigar benefícios obtidos com a DCEVizz. De modo geral, espera-se que DCEVizz contribua na compreensão e na eliminação de código morto, reduzindo a poluição do código e a relação negativa entre evolução e aumento da complexidade. Com a redução da poluição e da complexidade do código, manutenções podem ser realizadas de forma menos árdua, diminuindo seus custos.

Referências

Bastos, C; Júnior, P. A. P; Costa, H. Técnicas para Detecção de Código Morto: Uma Revisão Sistemática de Literatura. In: XII Simpósio Brasileiro de Sistemas de Informação. pp. 255-262. 2016.

Burd, L.; Rank, S. Using Automated Source Code Analysis for Software Evolution. In: IEEE International Workshop on Source Code Analysis and Manipulation. pp. 204-210. 2001.

Card, S. K.; Mackinlay, J. D.; Shneiderman, B. Readings in Information Visualization: Using Vision to Think. Morgan Kaufmann. 671p. 1999.

D'Ambros, M. Supporting Software Evolution Analysis with Historical Dependencies and Defect Information. In: International Conference on Software Maintenance. pp. 412-415. 2008.

Ducasse, S.; Lanza, M. The Class Blueprint: Visually Supporting the Understanding of Glasses. In: IEEE Transactions on Software Engineering. v.31, n.1, pp. 75-90. 2005.

Gold, N.; Mohan, A. A Framework for Understanding Conceptual Changes in Evolving Source Code. In: International Conference on Software Maintenance. pp. 431-439. 2003.

Lanza, M. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In: International Workshop on Principles of Software Evolution. pp. 37-42. 2001.

Lehman, M. M.; Belady, L. Program Evolution: Processes of Software Change. Academic Press. 538p. 1985.

Martin, R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall. Ed 1. 431p. 2008.

Minelli, R.; Lanza, M. SAMOA - Visual Software Analytics Platform for Mobile Applications. In: International Conference on Software Maintenance. pp. 476-479. 2013.

Novais, R. L.; Nunes, C.; Garcia, A.; Mendonça, M. Sourceminer Evolution: A Tool for Supporting Feature Evolution Comprehension. In: International Conference on Software Maintenance. pp. 508-511. 2013.

Romano, S.; Scanniello, G. DUM-Tool. In: International Conference on Software Maintenance and Evolution. pp. 339-341. 2015.

Scanniello, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In: Conference on Software Engineering and Advanced Applications. pp. 392-397. 2014.

Turo, D., Johnson, B. Improving the Visualization of Hierarchies with Treemaps: Design Issues and Experimentation. In: Conference on Visualization. pp. 124-131.1992.

Zhi, J. Ruhe, G. DEVis: A Tool for Visualizing Software Document Evolution. In: Working Conference on Software Visualization. pp. 1-4. 2013.

ARSENAL-GSD
Estimando confiança entre membros de uma equipe DGS

Guilherme A. M. da Cruz[1], **Elisa H. M. Huzita**[1], **Valéria D. Feltrim**[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
CEP 87.020-900 – Maringá – PR – Brasil

guilherme.maldonado.cruz@gmail.com, {elisahuzita,vfeltrim}@din.uem.br

Abstract. *The global software development aims at providing some benefits to software development, such as costs reduction, proximity to the market and faster products delivery. However, some challenges are added, especially when it is related to trust. Since trust impacts on several factors, trust information among team members can be used to suggest/monitor global software development teams to ensure their efficiency. We present an implementation of ARSENAL-GSD a framework to estimate and display the trust between members, and so to help project managers at the monitoring and selection of team members.*

Resumo. *O desenvolvimento global de software busca fornecer alguns benefícios ao desenvolvimento de software, tais como a redução dos custos, proximidade do mercado e agilidade na entrega de produtos. Contudo alguns desafios são adicionados, notadamente no que tange a confiança. Como a confiança impacta em diversos fatores, as informações de confiança entre os membros podem ser utilizadas para sugerir/monitorar equipes de desenvolvimento global de software a fim de garantir a sua eficiência. Neste trabalho apresentamos a implementação do framework ARSENAL-GSD que oferece o apoio necessário para estimar e visualizar a confiança entre membros auxiliando o monitoramento e escolha dos membros dessas equipes.*
Vídeo: `https://youtu.be/yE5DePYJ3yY`

1. Introdução

O desenvolvimento global de software (DGS) proporciona benefícios, tais como a redução dos custos, agilidade na entrega dos produtos, facilidade de encontrar mão de obra qualificada e proximidade do mercado [O'Conchuir et al. 2006]. Contudo, a distribuição geográfica também traz desafios, como divergências culturais, dificuldades técnicas e o estabelecimento da confiança entre os membros das equipes DGS.

A confiança é importante para as equipes de DGS uma vez que ela impacta no compartilhamento de conhecimento, coesão, e cooperação, entre outros aspectos que caracterizam equipes de alto desempenho [Siakas and Siakas 2008]. Portanto, informações referentes à confiança podem ser utilizadas para sugerir equipes para novos projetos/tarefas ou para monitorar projetos em andamento, a fim de garantir a eficiência das equipes.

Nesse contexto este trabalho apresenta uma implementação do *framework* ARSENAL-GSD [Cruz et al. 2016] para estimar e visualizar a confiança entre os membros de equipes de DGS. Nessa implementação, o ARSENAL-GSD estima valores de

confiança a partir da extração de indícios de confiança exibidos em interações entre membros de equipes de DGS por meio do GitHub e os exibe na forma de um grafo. Uma de suas principais características é o uso da análise de sentimentos para extrair parte dos indícios de confiança, o que garante automaticidade e subjetividade.

Além desta seção, na Seção 2, apresentamos os conceitos de DGS e confiança. Na Seção 3, apresentamos o ARSENAL-GSD. Na Seção 4, comparamos o ARSENAL-GSD com trabalhos relacionados. Por fim, na seção 5 apresentamos as conclusões e possíveis trabalhos futuros.

2. Revisão da Literatura

Esta seção apresenta os conceitos de DGS e confiança, que serviram de base para o desenvolvimento do ARSENAL-GSD.

2.1. Desenvolvimento Global de Software

Buscando a redução de custos, agilidade no desenvolvimento e acesso à mão de obra qualificada, as organizações passaram a utilizar equipes virtuais aliadas à terceirização, o chamado desenvolvimento global de software (DGS) [Jiménez and Piattini 2009]. O DGS é uma atividade colaborativa, que pode ser caracterizada por ter membros de diferentes culturas e organizações, separados pelo tempo e espaço, e que usam comunicação mediada por computador para colaborar [O'Conchuir et al. 2006, Sengupta et al. 2006]. Outros benefícios fornecidos pela utilização do DGS são: o desenvolvimento *follow-the-sun*, a modularização do trabalho entre os locais de desenvolvimentos, inovação e compartilhamento de práticas, e a proximidade do mercado.

Contudo, a distribuição geográfica dos membros acrescenta alguns desafios ao desenvolvimento de software no que ser refere a problemas estratégicos, culturais, de comunicação, de gerência de conhecimento e processos, e técnicos [Herbsleb and Moitra 2001, Sengupta et al. 2006]. Esses desafios também contribuem para que haja dificuldade em se estabelecer confiança entre os membros dessas equipes.

2.2. Confiança

Com base nas definições dadas em diversas áreas do conhecimento, Rusman et al. [2010, p.836, tradução nossa] definiram confiança como:

> Um estado psicológico positivo (cognitivo e emocional) do confiante em relação ao confiado, que compreende as expectativas positivas do confiante a respeito das intenções e comportamentos futuros do confiado, levando a uma vontade de expressar um comportamento de confiança em um contexto específico.

Essa definição apresenta uma das características da confiança, que é a sensibilidade ao contexto. Outras características da confiança são: dinâmica, não transitiva, propagativa, composta, subjetiva, assimétrica, autorreforço e sensível a eventos [Sherchan et al. 2013].

A confiança é particularmente importante para DGS, uma vez que a sua existência permite mitigar o comportamento oportunista, melhorar a comunicação, estabelecer a coesão, apoiar a criação de objetivos mútuos, facilitar a transferência de conhecimentos e recursos, aumentar a eficiência da equipe, melhorar a qualidade das saídas e alcançar objetivos mais facilmente [Siakas and Siakas 2008].

3. ARSENAL-GSD

Dada a importância da confiança para as equipes de DGS, nós desenvolvemos um *framework* automático para estimar a existência de confiança entre os membros dessas equipes. O *framework* foi chamado de ARSENAL-GSD (*Automatic tRust eStimator based on sENtiment anALysis for Global Software Development*) [Cruz et al. 2016] e suas principais características são:

(a) Utiliza sistemas de versionamento como fonte de dados;
(b) Utiliza indícios de confiança extraídos a partir dos dados do sistema de versionamento (comentários, estado do *commit* e perfil do usuário) e da análise de sentimentos;
(c) É automático;
(d) Preserva a subjetividade inerente à confiança;
(e) Atualiza os valores de indícios e de confiança com o tempo;
(f) Gera um grafo de relacionamentos;
(g) Gera um grafo de confiança com as estimativas de confiança;
(h) Tem foco na confiança interpessoal, mais especificamente, entre duas pessoas;
(i) Estima a confiança direta entre duas pessoas.

A Figura 1 apresenta o diagrama de componentes do *framework* ARSENAL-GSD, composto de quatro componentes, conforme descrito a seguir:

Graph Esse componente fornece ao *framework* as instâncias dos grafos de relacionamentos inicial e de confiança. Alterando esse componente, podemos armazenar os grafos como ontologias ou tabelas em bancos de dados, por exemplo.

VS Data Extractor Esse componente extrai dados do sistema de versionamento, tais como informações de perfil dos usuários e informações e conversas das *pull requests*. Além de extrair os dados, esse componente também gera o grafo de relacionamentos inicial.

Evidence Analyser Esse componente fornece classes que implementam a interface *EvidenceAnalyser*, representando uma técnica de extração de indício. Essas classes vão analisar os dados extraídos do sistema de versionamento e gerar valores que serão armazenados no grafo de relacionamentos e, posteriormente, utilizados para estimar a existência de confiança. Foram definidas seis técnicas para extração de indícios, são elas: mímica de vocabulário, atribuições, comunalidade, polaridade, *merges* e colaboração, que estão detalhadas em Cruz et al. [2016]. Para adicionar mais indícios ao *framework* é necessário somente uma nova implementação da interface *EvidenceAnalyser* para uma nova técnica de extração de indícios.

Trust Framework Esse é o componente principal e fornece meios de configurar e utilizar o *framework*. Esse componente é responsável por pegar os dados do `VS Data Extractor` e transmitir para o `Evidence Analyser`. A partir dos valores gerados pela extração de indícios, calcula-se a média ponderada dos mesmos e chega-se à estimativa de confiança. Na sequência, é gerado o grafo de confiança.

Note que providenciando novas implementações para o componente `VS Data Extractor` é possível estender o ARSENAL-GSD para outros sistemas de versionamento. Contudo, como cada sistema de versionamento pode disponibilizar dados diferentes, o componente `Evidence Analyser` está vinculado ao componente `VS Data Extractor`. Dessa forma, para suportar outro sistema de versionamento pode ser necessário substituir o componente `Evidence Analyser` por um que dê suporte ao

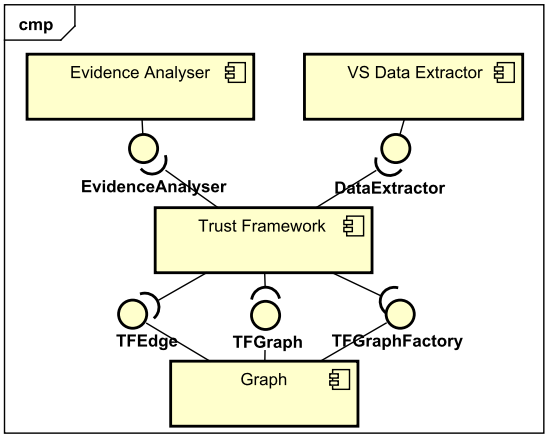


Figura 1. Diagrama de componentes para o *framework* ARSENAL-GSD.

novo sistema de versionamento. Também é possível estender o conjunto de indícios de confiança considerados fornecendo implementações da interface *EvidenceAnalyser*.

3.1. Implementação para o GitHub

Nós implementamos uma instância do ARSENAL-GSD para o sistema GitHub utilizando a linguagem Java. Para a implementação dos componentes e das técnicas de extração de indícios, foram utilizadas as seguintes APIs:

- Cytoscape.js[1]: Utilizada para exibir os grafos.
- JGraphT[2]: Utilizada para gerar os grafos de relacionamentos e confiança.
- GitHub Java API[3]: Utilizada para extrair os dados do GitHub.
- Lucene[4]: Utilizada para gerar o vetor de frequência de palavras necessário para a extração do indício mímica de vocabulário.
- Commons Math[5]: Fornece os métodos necessários para calcular a similaridade de cosseno, que é empregada na extração do indício mímica de vocabulário.
- Sentistrength [Thelwall et al. 2010]: Utilizada como API de análise de sentimentos. A partir da API obtemos os valores de polaridade para cada comentário, necessários para a extração do indício polaridade.
- AlchemyAPI[6]: Utilizada para extrair o alvo dos comentários (pessoas para as quais o comentário foi direcionado) a fim de calcular o valor de polaridade.
- Google Maps Geocoding API[7]: Utilizada para descobrir de qual país é o endereço fornecido por cada usuário. A localização é empregada na extração do indício comunalidade.

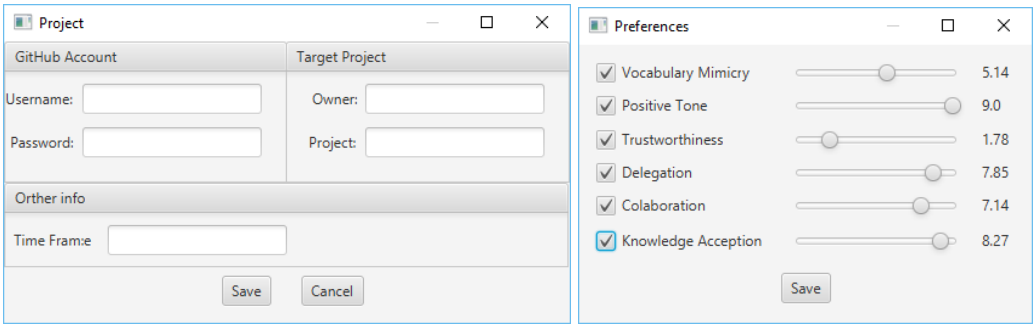
3.2. Funcionamento

O *framework* pode ser integrado a outras aplicações, mas também conta com uma interface gráfica simples, por meio da qual os usuários (gerentes de projeto, por exemplo)

[1]http://js.cytoscape.org/
[2]http://jgrapht.org/
[3]https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core
[4]https://lucene.apache.org/
[5]http://commons.apache.org/proper/commons-math/index.html
[6]http://www.alchemyapi.com/
[7]https://developers.google.com/maps/documentation/geocoding/intro

podem informar um projeto alvo do GitHub, as técnicas de extração de indícios e, a partir disso, estimar os valores de confiança entre os membros de um projeto.

Para ilustrar o uso da ferramenta, selecionamos o projeto real do GitHub, `objective-c-style-guide` do usuário `NYTimes`, considerando um período de 300 dias. A configuração do *framework* é realizada em duas telas: na primeira, mostrada na Figura 2(a), o usuário deve informar o projeto alvo (proprietário/repositório), uma conta do GitHub (é opcional, contudo, a API do GitHub permite uma menor quantidade de requisições sem uma conta) e a janela de tempo que deve ser considerado. Na segunda tela, mostrada na Figura 2(b), o usuário deve informar quais técnicas de extração de indícios devem ser utilizadas e qual o peso de cada uma.



(a) Configuração do projeto. (b) Configuração das técnicas de extração.

Figura 2. Telas de configuração.

A Figura 3 apresenta a tela principal do *framework*, na qual temos seis botões:

- *Compute*: Executa o *framework* e exibe os participantes envolvidos.
- *Relations*: Exibe as arestas de relacionamentos entre os membros, que correspondem às *pull requests* nas quais houve interação.
- *Evidences*: Exibe as arestas com os valores dos indícios extraídos.
- *Trust*: Exibe as arestas com as estimativas de confiança entre os membros.
- *Compare*: Exibe um grafo salvo, permitindo a comparação com o grafo atual.
- *Reset*: Desfaz a comparação, voltando a exibir somente o grafo gerado.

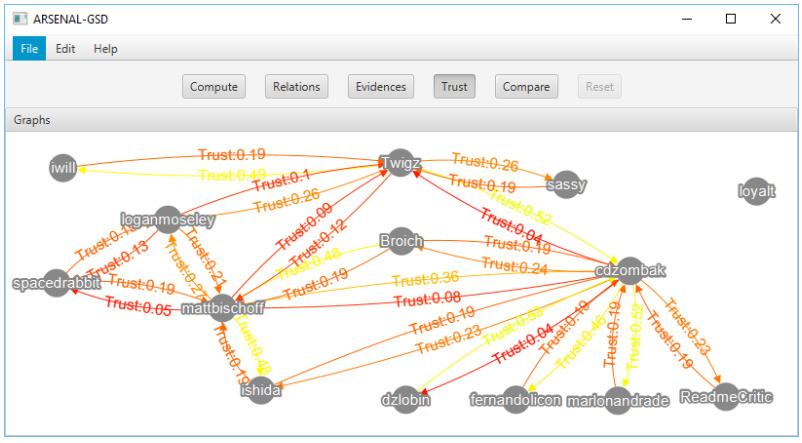


Figura 3. Tela principal exibindo as estimativas para um projeto.

Como podemos ver na Figura 3, o grafo de confiança exibido possui arestas coloridas. As cores vão de vermelho (0) a verde (1). Quanto mais próximo de verde (1), maior a chance de existir confiança entre os membros. A utilização desse gradiente de cores permite que os usuários do *framework* consigam identificar, com facilidade, as relações de maior confiança e, a partir disso, possam escolher membros para um novo projeto/tarefa. No exemplo da Figura 3, vemos que o membro **cdzombak** e os membros adjacentes a ele possuem as arestas com cores mais próximas de verde, mostrando que existe uma maior confiança entre eles em comparação aos demais membros representados no grafo; evidenciando, assim, candidatos que constituem em uma boa opção para uma nova equipe.

O grafo gerado para um projeto pode ser salvo e utilizado, posteriormente, para comparação. Para efeito de ilustração, salvamos um grafo considerando 150 dias e o utilizamos para comparação com o grafo de 300 dias. A Figura 4 mostra a tela de comparação, na qual os dois grafos são colocados lado a lado, permitindo ao usuário do *framework* analisar as diferenças entre eles. Essa funcionalidade é ideal para comparar a evolução da confiança com o passar do tempo. Por exemplo, podemos gerar um grafo a cada 30 dias e comparar com o anterior, para identificar quedas na confiança da equipe. Uma vez identificadas essa quedas, medidas corretivas podem ser tomadas pelo gerente de projetos para recuperar a confiança e, assim, manter e/ou melhorar a eficiência da equipe.

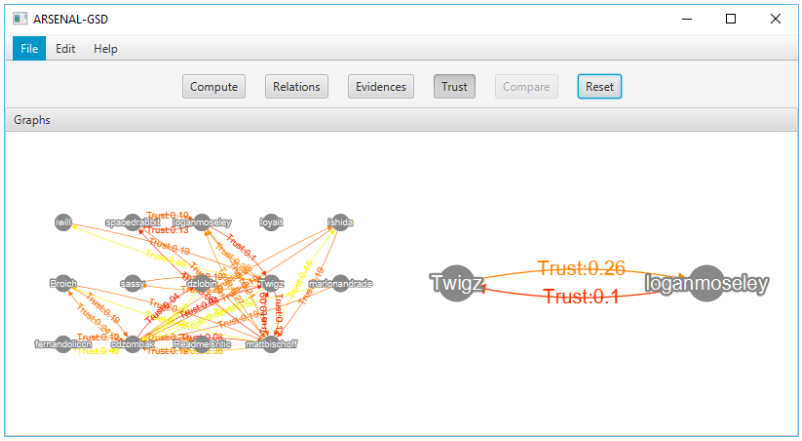


Figura 4. Tela de comparação de grafos.

3.3. Validação

Para validar o *framework* realizamos uma *survey* com 13 participantes (estudantes, pesquisadores e profissionais da indústria) contendo perguntas a respeito da importância dos indícios de confiança considerados (respostas com valores entre 0 e 9) e da validade das técnicas de extração de indícios implementadas (as respostas possíveis eram: 3-sim, prefeita; 2-sim, boa o suficiente; 1-regular e 0-não).

Para as perguntas relativas à importância dos indícios, a média das respostas para cada indício foi superior a cinco. Desse modo, assumimos que os indícios considerados são relevantes. Para as respostas referentes às técnicas utilizadas para a extração de indícios, moda e mediana foram igual a dois. Assim, consideramos que as técnicas empregadas, apesar de não serem perfeitas, são válidas para extrair os indícios propostos.

4. Trabalhos Relacionados

Fan et al. [2011] e Skopik et al. [2009] também propuseram *frameworks* para estimar a confiança. A proposta de Fan et al. [2011] utiliza valores de reputação e colaboração entre os membros da equipe para fazer a estimativa. Esses valores são obtidos por meio de avaliações fornecidas pelos próprios membros. Depois de computados, os valores de reputação e colaboração formam um ponto em uma matriz representando o nível de confiança do membro na equipe.

O trabalho de Skopik et al. [2009] busca determinar a confiança entre pessoas e serviços de forma automática a partir da quantidade de interações bem-sucedidas, determinadas por um conjunto de métricas, em relação ao total de interações. Assim como em nossa proposta, os valores de confiança calculados são apresentados em um grafo.

A principal diferença entre o ARSENAL-GSD e esses trabalhos está na união da automaticidade e da subjetividade em um mesmo *framework*, características presentes de forma isolada nos trabalhos de Fan et al. [2011] (sub.) e Skopik et al. [2009] (aut.).

A análise de sentimentos também foi usada por O'Donovan et al. [2007] na estimativa de confiança. Os autores analisaram os comentários dos *feedbacks* do Ebay para medir a confiança dos vendedores. Foram empregadas técnicas de análise de sentimentos para determinar a polaridade dos comentários e, a partir da polaridade, se gerava a confiança granular e a confiança interpessoal.

5. Conclusão

A eficiência de uma equipe de DGS está diretamente ligada à confiança entre os membros da equipe. A existência de confiança aumenta a comunicação, facilita a cooperação, coordenação e o compartilhamento de conhecimento e informações, que melhoram a qualidade dos produtos gerados. Motivados pela importância da confiança para essas equipes, apresentamos uma implementação do *framework* ARSENAL-GSD para o GitHub e provemos uma interface gráfica para a configuração e visualização dos grafos de relacionamentos e de confiança.

Gerentes de equipes DGS podem se beneficiar da ferramenta tanto por meio do grafo de relacionamentos, quanto do grafo de confiança. O grafo de relacionamentos permite ao gerente observar quais membros são mais ativos e interagem mais dentro da equipe. Esses membros podem ser uma boa opção para coordenar equipes ou inteirar novos membros sobre o andamento do projeto. A coloração do grafo de confiança em gradiente facilita encontrar as relações de maior confiança, que por sua vez, auxilia o gerente de projetos na constituição de uma equipe com maior nível de confiança. A funcionalidade de comparação de grafos permite monitorar variações no nível de confiança em equipes existentes, permitindo que ações corretivas possam ser tomadas quando houver uma queda no nível de confiança da equipe.

Como um trabalho futuro, pretendemos evoluir a funcionalidade de comparação de grafos para realizá-la automaticamente, mostrando os pontos em que houve queda da confiança e quais das técnicas de extração de indício contribuíram para a queda.

6. Agradecimentos

Agradecemos a CAPES pelo apoio financeiro.

Referências

Cruz, G., Huzita, E., and Feltrim, V. (2016). Estimating trust in virtual teams - a framework based on sentiment analysis. In *Proceedings of the 18th International Conference on Enterprise Information Systems (ICEIS 2016)*, volume 1, pages 464–471.

Fan, Z.-P., Suo, W.-L., Feng, B., and Liu, Y. (2011). Trust estimation in a virtual team: A decision support method. *Expert Systems with Applications*, 38(8):10240–10251.

Herbsleb, J. D. and Moitra, D. (2001). Global software development. *IEEE Software*, 18(2):16–20.

Jiménez, M. and Piattini, M. (2009). Problems and solutions in distributed software development: A systematic review. In Berkling, K., Joseph, M., Meyer, B., and Nordio, M., editors, *Software Engineering Approaches for Offshore and Outsourced Development*, number 16 in Lecture Notes in Business Information Processing, pages 107–125. Springer Berlin Heidelberg.

O'Conchuir, E., Holmstrom, H., Agerfalk, P., and Fitzgerald, B. (2006). Exploring the assumed benefits of global software development. In *International Conference on Global Software Engineering, 2006. ICGSE '06*, pages 159–168.

O'Donovan, J., Smyth, B., Evrim, V., and McLeod, D. (2007). Extracting and visualizing trust relationships from online auction feedback comments. In *20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2826–2831, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Rusman, E., van Bruggen, J., Sloep, P., and Koper, R. (2010). Fostering trust in virtual project teams: Towards a design framework grounded in a TrustWorthiness ANtecedents (TWAN) schema. *International Journal of Human-Computer Studies*, 68(11):834–850.

Sengupta, B., Chandra, S., and Sinha, V. (2006). A research agenda for distributed software development. In *International Conference on Software Engineering*, pages 731–740. ACM.

Sherchan, W., Nepal, S., and Paris, C. (2013). A survey of trust in social networks. *ACM Computing Surveys (CSUR)*, 45(4):47:1–47:33.

Siakas, K. V. and Siakas, E. (2008). The need for trust relationships to enable successful virtual team collaboration in software outsourcing. *International Journal of Technology, Policy and Management*, 8(1):59–75.

Skopik, F., Truong, H. L., and Dustdar, S. (2009). Viete - enabling trust emergence in service-oriented collaborative environments. In *WEBIST 2009 - Fifth International Conference on Web Information Systems and Technologies*, pages 471–478.

Thelwall, M., Buckley, K., Paltoglou, G., Cai, D., and Kappas, A. (2010). Sentiment in short strength detection informal text. *Journal of the American Society for Information Science and Technology*, 61(12):2544–2558.

EVOWAVE – A Multiple Domain Tool for Software Evolution Visualization

Rodrigo Magnavita[1], **Renato Novais**[1,2], **Manoel Mendonça**[1]

[1]Fraunhofer Project Center at UFBA
Salvador, BA.

[2]Instituto Federal da Bahia
Salvador, BA.

{rodrigo.magnavita, manoel.mendonca}@ufba.br, renato@ifba.edu.br

Abstract. *To understand all the data produced while software evolves and recovery valuable information is a challenge. Software Evolution Visualization (SEV) can be used to this end. However, this is not trivial, since SEV has to handle different software entities and attributes, and still deals with the temporal dimension of evolution. SEV tools are generally built for a specific domain of software engineering, focusing mainly only on presenting an overview of the software development, without focusing on the detail. However, most software development activities require: combine overview and detailed information of different domains. This work presents a tool, called EVOWAVE, which realizes a novel visual metaphor. It is able to address multiple domains and to comprehensively visualize large amount of data in a overview and detailed way. In this paper, we show the EVOWAVE applicability through two different domains. Video - http://wiki.ifba.edu.br/evowave*

1. Introduction

Software evolution has been highlighted as one of the most important topics in software engineering [Novais et al. 2013]. It has very complex activities, generating a huge amount of data. The challenge is to have methods, processes and techniques to support the comprehension of all the data, recovering valuable information to perform the tasks in a successful way.

Software visualization has been used as one way to deal with software comprehension activities. It helps people to understand software through visual elements, reducing complexity to analyze the large amount of data generated during the software evolution [Diehl 2007]. Some examples of what this data can be are: software metrics, stakeholders, bugs, and features. To build visual metaphors that effectively represent the time dimension with all the data related to software evolution is a challenge in the field of software evolution visualization (SEV). In the context of SEV, researchers have taken different approaches. Some present the big picture of the software, providing an overview of the entire software history [Kuhn et al. 2010][Lungu 2008], while others show snapshots of the software evolution in detail [D'Ambros et al. 2009][Novais et al. 2011][Novais et al. 2012]. They are both important because each approach fits better to specific software evolution tasks. An important issue in the area is to understand how to combine both approaches in a practical

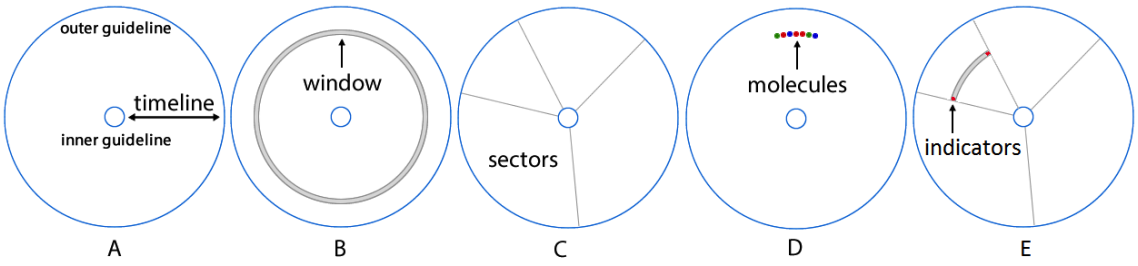


Figure 1. The EVOWAVE concepts

and useful way, so that users can really take advantage of the proposed visualizations [Novais et al. 2013].

In this work, we present a new SEV tool, called EVOWAVE. It realizes a novel visualization metaphor [Magnavita et al. 2015] that is able to visualize different types of data generated in software evolution using both overview and detail approaches. EVOWAVE can represent a huge number of events at a glance. Several mechanisms of interactions allow the user to explore the visualization in detail. This open source tool can be applied to different software engineering tasks and domains. To validate its benefits, we conducted exploratory studies using EVOWAVE in three different software engineering domains: software collaboration [Magnavita et al. 2015], library dependency, and logical coupling.

This paper is organized as follow. Section 2 presents the EVOWAVE tool, highlighting its main features. Section 3 shows its use in two different domains. Section 4 presents related work. Finally, Section 5 concludes this work.

2. EVOWAVE

EVOWAVE is a new visualization tool that enriches the analysis capabilities of software evolution. It is inspired on concentric waves with the same origin point in a container seen from the top. This section presents the concepts related to the proposed metaphor, which makes EVOWAVE. Later, we explain how those concepts can be mapped to software properties and the tool's aspects of implementation.

2.1. EVOWAVE Concepts

Figure 1 presents the EVOWAVE concepts, which are explained below.

Layout. EVOWAVE has a circular layout with two circular guidelines (inner and outer), as shown in Figure 1-A. They represent a software life cycle period between two selected dates. This period, named timeline (Figure 1-A), gives an overview of the software history. It is comprised by a series of short periods with the same periodicity (e.g., ten days, two hours, one month). The periodicity may differ between visualizations according to the size of the display available and users' configuration. To give some orientation to the path between the guidelines, the inner can be mapped to the oldest date and the outer to the newest date, or vice versa.

Windows. A window is a group of consecutive short periods (Figure 1-B). It is circular in shape and its length depends on the number of grouped periods. It can be used

to compare a subset of short periods, making it possible to carry out a detailed analysis regarding the overall context.

Sectors. A sector is a visual element drawn between the two circular guidelines according to its angle (Figure 1-C). It may have different angles. This concept is used to group events that share some characteristic (e.g., classes of the same packages). Considering, for example, a sector representing a package, it is possible to interact and navigate getting detail on demand about the inner packages.

Molecules. Molecules are depicted as circular elements inside sectors and windows (Figure 1-D). Each molecule has an event associated to it. When we have more molecules than the display size limits, we gathered and drawn them as a quadrilateral polygon that fills the region where the molecules are. Molecules can represent any change on software history, such as file changes, team changes or bug reports.

Indicators. The number of molecules indicator is drawn as a rectangle located in the frontier of the sectors for each window (Figure 1-E). Its color varies from red to blue, where the reddest indicator has the largest number of molecules and the bluest has lowest number of molecules. The indicator can refer to the local sector or to all sectors. If set to local, its color will take in consideration the other windows inside the sector. Otherwise, if set to all sectors (global), its color will take in consideration all windows present in the visualization.

2.2. Mapping software properties

EVOWAVE concepts define how the metaphor organizes and displays events, which occurred during any general data history. In this sense, the EVOWAVE metaphor is able to represent software evolution, by mapping its visual elements to software history attributes. It is important to take into account that each mapping will give different information and should be chosen according to the software development tasks at hand. Find below the EVOWAVE characteristics (visual attributes) that can be mapped to software history attributes (real). Figure 2 shows two examples of EVOWAVE. On the left side, it shows a dependency usage of FindBugs project. For sake of understanding, this example is decorated (mockup) with labels pointing to the main concepts of the metaphor. On the right side, it shows a logical coupling of ArgoUML project (this example will be explored on the next section).

Timeline. The timeline defines the period of analysis through two dates. We can map two software versions and analyze what happened between them. If we map the first version to the inner guideline, and the last version to the outer guideline, the history of the software is portrayed from the center to the periphery.

The Pooler of a Sector. A pooler defines how the events will be grouped. The software property chosen to be the pooler has to categorize the events. Events within the same category will be in the same sector. Some examples of software properties that can be mapped to a pooler are: the package of a changed class event; the file type of a changed file event; and the author of a bug report event.

The Splitter of a Sector. The splitter defines how the hierarchy of the pooler's property will be created. The pooler's property usually has some delimiter that can be point out to be the splitter's property. The splitter needs to be part of the pooler in order

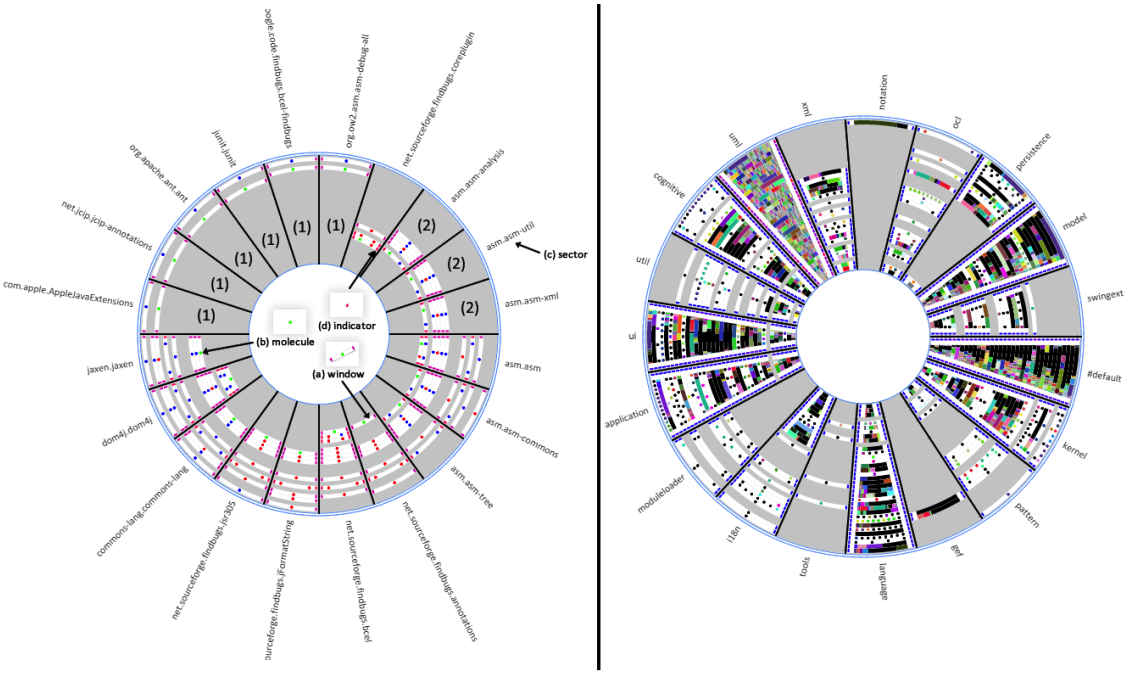


Figure 2. Left - The dependency usage of FindBugs project / Right - EVOWAVE visualization with all changes separated by modules from 2003 to 2005 with the "uml" package in focus.

to split it into different levels. For example, the slash character could be used as a splitter for the file path of a changed file. It is part of the pooler's property and will divide it into many folders where each one has a different level in the hierarchy.

The Angle of a Sector. The angle defines how much of some software property the sector has relatively to the others. If the pooler is a package of a changed class, the angle could be the increase in the package complexity, for example. In this case, all the events with increased complexities are summed up for each package. The highest complexity is mapped to the largest sector angle.

The Color of a Molecule. The user can select a color to map software properties as an event categorization or a numerical property range. An example of the event categorization are the authors of a code change, or of a bug report, where each author could have a different color associated to them. For numerical property range we may consider how much a Java class complexity grew or shrank. In this case, we can select two specific colors to decorate the changed file with the most increased complexity and the most decreased complexity. Any event between these two ones will have its color interpolated. When there are too many molecules to display, a quadrilateral polygon is drawn and its color can be associated to the number of molecules in it or to the proportion of each color.

It is important to remark that EVOWAVE is highly configurable. In this sense, the user can select any visual attribute in the available set, and map to real attributes he/she has. In our opinion, the task at hand will guide this mapping.

2.3. Aspects of Implementation

Evowave architecture is designed to be simple and extensible. The client side uses a well-known Javascript framework called ProcessingJS [ProcessingJS 2016], which is used to draw the visualization. The metadata used by it is defined in a key-value notation called JSON (Javascript Object Notation). The JSON has all the data needed by the visualization to depict the data following the metaphor's concept. This data is collected by the server according to the period defined. The client will never ask extra data from the server as long as the client does not change the period of analysis. The nature of this tool is to consume services that returns all the information needed to perform an action. Thus, in the server's side, a servlet that fulfill the RESTful architecture was used. Every service provided by the server returns the data according to the visualization metadata. More details on the algorithms used can be found in [Magnavita 2016].

3. Feasibility Studies in a Nutshell

EVOWAVE is designed to be a multiple domain metaphor for software evolution visualization. In order to validate its goal, we conducted three studies in different evaluation scenarios. Each study exercises the EVOWAVE metaphor in a different software evolution domain: Software Collaboration domain [Magnavita et al. 2015]; Library Dependency domain; and Logical Coupling domain. Tasks related to each one of those domains were performed using the EVOWAVE metaphor with different data from different repositories. Due to space restriction, in this section we show a brief overview of the last two studies. In [Magnavita 2016], the reader can see all the details of those studies.

3.1. Library Dependency domain

We conducted an exploratory study to evaluate the use of EVOWAVE in the analysis of the evolution of how the dependency relation between a system and its dependencies evolves. Using the GQM paradigm [Basili and Rombach 1988], the goal of this study was: **To analyze** the EVOWAVE metaphor; **With the purpose of** characterize; **Regarding** the dependency relationship evolution between a system and its dependencies; **From the point of view** of EVOWAVE researchers; **In the context of** software comprehension tasks designed by Kula [Kula et al. 2014] in FindBugs - a real world open source system.

An HTML parser was developed to read page by page of the Maven repository to extract the dependency data from the FindBugs system. During the analysis of 196,329 HTML pages (5,5GB), the parser extracted 15 versions of the FindBugs system, 294 dependencies from the FindBugs system and 162,510 system versions that use at least one of those dependencies. Figure 2-left shows one example of EVOWAVE visualization for this study. We setup the metaphor as follows: **The period of analysis is** from February 11, 2003 to November 15, 2015; **Sectors** are the dependencies from the FindBugs system. Each dependency has sectors inside, which represent dependency versions; **Windows** represent six months; **Molecules** represents the use of only one dependency by FindBugs system itself or another system; **Molecule Colors** represent if the system that is using the dependency is adopting (green), remaining (blue), or updating (red) this dependency.

Using EVOWAVE, we were able to perform the tasks by [Kula et al. 2014]. They are: *Understand the regularity of system dependency changes*; *Understand what important structural dependency events have occurred*; *Discover the current "attractiveness" of any library version*; *Discover if newer releases are viable candidates for updating*.

Figure 2-left displays the visualization filtered only for the dependency usages by the FindBugs system. To *understand the regularity of changes in the system dependency*, the software engineer needs to look for molecules changes in the sectors. The "com.apple.Apple JavaExtensions", "net.jcip.jcip-annotations", "org.apache.ant.ant", "junit.junit", "com. google.code.findbugs.bcel-findbugs", and "org.ow2.asm.asm-debug-all" dependencies are new for the FindBugs system and were never updated (they are represented as (1) in Figure 2-left). We can reach this conclusion because the molecules in the sectors related to those dependencies are presented on the edge of the visualization and there are no red molecules in it. Unlike those dependencies, the "net.sourceforge.findbugs.annotations" dependency is old and highly changed. The FindBugs system used this dependency from the first version until the middle of 2014. The dependency was updated almost every time a new version of the FindBugs system was released. This behavior represents an indication of high coupling between those systems. Another important behavior to notice is that this dependency was not used in some versions. For example, the next window after the window that holds the green molecule for this dependency is empty. But there were new versions released during this period as we can see in the sector "net.sourceforge.findbugs.bcel". The "net.sourceforge.findbugs.annotations" dependency was used after this window again. This behavior represents that the features provided by this dependency might be replaced by another library. The EVOWAVE visualization, as in Figure 2-left (2), can also be used to address this task by looking for changes in the dependencies. The dependencies "asm.asm-analysis", "asm.asm-util" and "asm.asm-xml" were removed from the project at the same time. This may imply that they were used for some common feature that is using a different library or were removed.

3.2. Logical Coupling domain

We conducted an exploratory study to validate the use of EVOWAVE to analyze the logical coupling between system modules during the software development process. Using the GQM paradigm, the goal of this study was: **To analyze** the EVOWAVE metaphor; **With the purpose of** characterize; **Regarding** logical coupling evolution between software modules; **From the point of view** of EVOWAVE researchers; **In the context of** a retrospective analysis made by Ambros [D'Ambros et al. 2009], in ArgoUML - a real world open source system.

A parser was developed to read Subversion log files and extract the files that were changed along with a file in the "org.argouml.uml" package. For each commit we extract the following data: the changed file and its package, when the change occurred, and the commit's id. Figure 2-right shows one example of EVOWAVE visualization for this study. We setup the metaphor as follows: **The period of analysis** from September 4, 2000 to January 11, 2015; **Sectors** are software modules; **Windows** represent six months; **Molecules** represent changes in a java file; **Molecule Colors** represent the commit of the change. The commits in black made no changes in the selected module, if one was selected.

Using EVOWAVE, we were able to make a retrospective analysis of the logical coupling evolution in the ArgoUML system. To start investigating in more details, we need to analyze the logic coupling of the "uml" package with other system modules. Figure 2-right illustrates the visualization to analyze the logical coupling of the "uml"

package with other packages. The package "moduleloader" has a low logical coupling with the "uml" package. Nevertheless, in the fifth first months for the "moduleloader" package, there were three changes in the same commit that changed almost all modules. Looking deeper into the comment of this commit and into the changes, we identified they were related to the copyright style and impacted 1,065 files. In terms of source code, they have no impact but let us know that probably every single file must change when there is a new copyright information. The package "ocl" is highly coupled with the "uml" package. This is observed because there is almost no black color in the "ocl" sector. Among the changes, there are two commits highlights in the forth newer window: the green and red colors. The commit related to the green color is the copyright change that we reported while analyzing the "moduleloader" package. The commit represented by the red color seems to impact many packages. While analyzing the comment and the changes from this commit we identified a major change in the system. The class responsible to be the facade for all communications with the "model" package, changed its signature from "ModelFacade" to "Model.getFacade()". This is a big change in the system because it breaks the signature used in many packages (e.g., ocl, persistence, uml). The logical coupling between the "ocl" package and the "uml" package for this commit is a consequence of their coupling with the model package.

Those two feasibility studies show the potential of EVOWAVE for visually analyze large amount of data using overview and detail approaches in order to support the understanding of software engineering tasks in different domains.

4. Related Work

In our literature review, we found two stand out related work for being similar to this work. They are the same we used as base line in two last exploratory studies. They use a similar layout but their proposal focus on different tasks for a unique domain.

In [D'Ambros et al. 2009], the authors proposed a visualization-based approach, named Evolution Radar, which integrates logical coupling information at different levels of abstraction. It shows the dependencies between a module in focus and all the other modules of a system. With that visualization, it is possible to track dependency changes detecting files with a strong logical coupling with respect to the last period of time, and then, analyze the coupling in the past, allowing us to distinguish between persistent and recent logical couplings. In [Kula et al. 2014], the authors proposed to visualize how the dependency relationship in a system and its dependencies evolves from two perspectives. The first one uses the same radial layout but with different concepts, and includes the use of heat-map to provide visual clues about the change in the library dependencies along with the system's release history. The second one uses statistic graphics to create a time-series visualization that shows the diffusion of users across the different versions of a library.

5. Final Remarks

The use of a single metaphor to represent data from different domains is a novel approach. In this work we presented EVOWAVE tool: a multiple domain software evolution visualization metaphor. We explain its concepts and how one can map real attributes to visual attributes. The tool is very flexible and can easy portray software history data. In other to

show its usefulness, we briefly presented two exploratory studies we conducted to evaluate EVOWAVE in different software engineering domains. We believe EVOWAVE is a very promising tool. As future work, we plan to evolve the tool, to overcome its current limitation as, for example, configuration mechanisms for data entry. Currently, we still do it by changing the code to point to a data set, and to do the mapping. We also plan to add more mechanisms of interaction, and validate it with other experimental studies, from the point of view of real users.

References

[Basili and Rombach 1988] Basili, V. R. and Rombach, H. D. (1988). The tame project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Eng.*, 14(6):758–773.

[D'Ambros et al. 2009] D'Ambros, M., Lanza, M., and Lungu, M. (2009). Visualizing co-change information with the evolution radar. *IEEE Trans. Softw. Eng.*, 35(5):720–735.

[Diehl 2007] Diehl, S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Kuhn et al. 2010] Kuhn, A., Erni, D., Loretan, P., and Nierstrasz, O. (2010). Software cartography: thematic software visualization with consistent layout. *J. Softw. Maint. Evol.*, 22(3):191–210.

[Kula et al. 2014] Kula, R., De Roover, C., German, D., Ishio, T., and Inoue, K. (2014). Visualizing the evolution of systems and their library dependencies. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 127–136.

[Lungu 2008] Lungu, M. (2008). Towards reverse engineering software ecosystems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 428 –431.

[Magnavita 2016] Magnavita, R. (2016). *EVOWAVE: A Multiple Domain Metaphor for Software Evolution Visualization*. Dissertation, Universidade Federal da Bahia.

[Magnavita et al. 2015] Magnavita, R., Novais, R., and Mendonça, M. (2015). Using evowave to analyze software evolution. In *Proceedings of the 17th International Conference on Enterprise Information Systems*, pages 126–136.

[Novais et al. 2011] Novais, R., Lima, C., de F Carneiro, G., Paulo, R., and Mendonça, M. (2011). An interactive differential and temporal approach to visually analyze software evolution. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1–4.

[Novais et al. 2012] Novais, R., Nunes, C., Lima, C., Cirilo, E., Dantas, F., Garcia, A., and Mendonca, M. (2012). On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1044–1053.

[Novais et al. 2013] Novais, R. L., Torres, A., Mendes, T. S., Mendonça, M., and Zazworka, N. (2013). Software evolution visualization: A systematic mapping study. *Inf. Softw. Technol.*, 55(11):1860–1883.

[ProcessingJS 2016] ProcessingJS (2016). A port of the processing visualization language. Retrieved from http://processingjs.org/.

ContextLongMethod: Uma Ferramenta Sensível à Arquitetura para Detecção de Métodos Longos

Cleverton Santos[1], Marcos Dósea[1, 2], Cláudio Sant'Anna[2]

[1] Departamento de Sistemas de Informação
Universidade Federal de Sergipe (UFS) – Itabaiana, SE – Brasil

[2] Departamento de Ciência da Computação
Universidade Federal da Bahia (UFBA) – Salvador, BA – Brasil

clevertonmaggot@gmail.com, dosea@ufs.br, santanna@dcc.ufba.br

Resumo. *Métodos longos no código em desenvolvimento podem ser sintomas da erosão do design. Para detectar essa anomalia de código, as ferramentas utilizam técnicas baseadas principalmente na análise da métrica de número de linhas de código. Entretanto, utilizar essa métrica sem considerar o contexto arquitetural da classe gera muitos falsos positivos e falsos negativos. Esse artigo apresenta a ferramenta ContextLongMethod que considera o contexto arquitetural da classe para recomendar potenciais métodos longos para o desenvolvedor. Avaliamos a ferramenta proposta e os resultados iniciais indicam maior acurácia em relação às ferramentas existentes.*

Abstract. *Long Methods can be a symptom of design erosion in the code being developed. To detect this design anomaly, existing tools mainly use techniques based on the analysis of the number of lines of code metric. However, using this metric without considering the class architectural context generates many false positives and false negatives. This work presents the ContextLongMethod tool, which considers class architectural context to recommend potential long methods to developers. We evaluated the proposed tool, and the initial results show higher accuracy in comparison with existing tools.*

Vídeo da Ferramenta: *https://youtu.be/FCx2yKqcrRk*

1. Introdução

A manifestação progressiva de métodos longos no código incrementa o processo de erosão do *design* do software. A erosão do *design* é um processo inevitável, mas boas práticas de desenvolvido incrementam a longevidade do sistema [van Gurp & Bosch 2002]. Segundo Fontana et al. (2013) os métodos longos estão entre as anomalias de código mais comuns independente do domínio da aplicação.

Para evitar a ocorrência de métodos longos e auxiliar a manutenção da qualidade, a revisão de código é uma prática comum utilizada pelas equipes de desenvolvimento. Recentemente várias abordagens automáticas para revisão do código vêm sendo propostas [Marinescu 2004; Arcoverde et al. 2012; Palomba et al. 2013]. A abordagem tradicional usada pelas ferramentas para detecção de métodos longos é baseada na definição de valores limiares para a métrica de número de linhas de código por método (SLOC/Método). Esse valor é configurado antes da análise do código e quando ultrapassado as ferramentas indicam os métodos que violam essa regra de *design*.

Entretanto, utilizar valores limiares genéricos para analisar todas as classes da aplicação acaba desconsiderando informações contextuais da arquitetura. Por exemplo, em sistemas que seguem o padrão arquitetural MVC, será que há diferença significativa entre o número médio de linhas de código em classes com papel de *Controller* e outras com papel de *Model*? E entre classes com o mesmo papel de *Controller* em sistemas distintos? Se isso ocorrer, a generalização pode impedir a detecção de anomalias de *design* ou ainda detectar anomalias não relevantes. Zhang et al. (2013) mostram evidências que informações do contexto da aplicação como o domínio, número de mudanças, tamanho e idade do sistema devem ser consideradas na utilização de métricas. Isso evitaria um grande número de falsos positivos e falsos negativos enviados para os desenvolvedores que podem se desmotivar em utilizar abordagens automáticas.

Outro problema é que muitas ferramentas precisam ser executadas manualmente pelo desenvolvedor. Essa tarefa pode ser esquecida ou ainda executada tardiamente no processo de codificação. Postergar a reparação dessas anomalias para fase final do processo de codificação pode levar a falta de motivação e tempo para consertar os problemas detectados.

Nesse contexto esse trabalho apresenta *ContextLongMethod*, uma ferramenta que recomenda candidatos a métodos longos a partir de informações extraídas do *design* de um sistema referência. O sistema referência de design pode ser uma versão anterior revisada do mesmo sistema ou ainda outro sistema que possua decisões de *design* semelhantes ao sistema que será avaliado. O contexto utilizado pela ferramenta é o interesse arquitetural da classe que considera: (1) o design do código da aplicação e (2) o papel arquitetural da classe. Os detalhamentos sobre a técnica implementada pela ferramenta e sobre a avaliação realizada podem ser encontrados em [Dósea et al. 2016].

O restante desse artigo está organizado como segue: a Seção 2 descreve o funcionamento da ferramenta *ContextLongMethod*, mostra um exemplo de utilização, sua arquitetura e uma avaliação inicial realizada comparando-a com outras ferramentas. Na Seção 3 são discutidas as ferramentas relacionadas e na Seção 4 as considerações finais e trabalhos futuros.

2. A Ferramenta *ContextLongMethod*

A ferramenta *ContextLongMethod* é um sistema de recomendação [Robillard et al. 2010] que extrai conhecimento do *design* de um sistema referência e utiliza-o para recomendar candidatos a métodos longos para o desenvolvedor. A ferramenta foi desenvolvida como um *plug-in* para o ambiente de desenvolvimento Eclipse e disponibiliza três abordagens para identificação de métodos longos:

a) Valor limiar genérico.
b) Valor limiar genérico extraído do *design* de uma aplicação referência.
c) Valores limiares para cada interesse arquitetural extraídos do *design* de uma aplicação referência.

A primeira abordagem considera um valor limiar genérico que pode ser configurado e é utilizado para avaliar todas as classes. Essa é a técnica mais utilizada pelas ferramentas existentes. A segunda abordagem utiliza um valore limiar extraído de

um sistema referência. Essa abordagem visa extrair valores limiares considerando um *design* mais próximo do sistema que será avaliado. A ferramenta calcula o valor da mediana e utiliza como valor limiar máximo o quartil 75 da lista de valores ordenada da métrica SLOC/Método. A mediana foi utilizada por ser uma estatística menos suscetível a variações no conjunto de valores e o valor do quartil pode ser configurado na ferramenta. Finalmente a última abordagem identifica valores limiares para cada interesse arquitetural identificado no sistema referência e também utiliza o quartil 75 como valore limiar padrão. Essa última abordagem é detalhada a seguir.

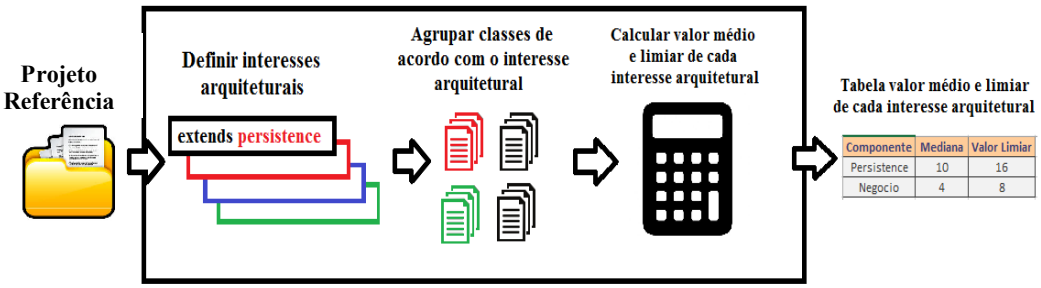


Figura 1. Definição de Valores Limiares a partir de um Sistema Referência.

A Figura 1 ilustra o processo referente à definição dos interesses arquiteturais do sistema referência e ao cálculo dos valores limiares para cada interesse. A abordagem é dividida em três etapas: (i) identificar o principal interesse arquitetural de cada classe do sistema referência (ii) agrupar classes de acordo com o interesse arquitetural e (iii) calcular valores limiares de cada interesse arquitetural. A saída desse processo é uma tabela contendo todos os interesses arquiteturais identificados e seus respectivos valores limiares.

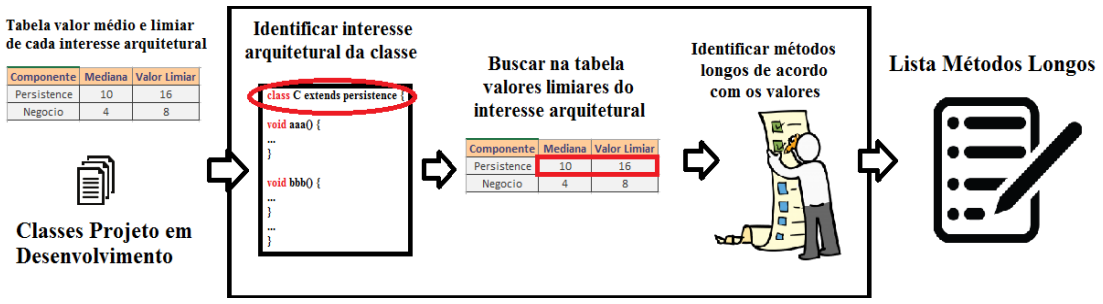


Figura 2. Processo de identificação de métodos longos com base em interesses arquiteturais.

A Figura 2 ilustra o processo de identificação de métodos longos considerando o interesse arquitetural. Para entrada do processo é utilizada a tabela que contém os valores limiares de cada interesse arquitetural calculado na primeira etapa. O processo é dividido em três tarefas (i) identificar interesse arquitetural da classe, (ii) buscar na tabela valores limiares do interesse arquitetural correspondente e (iii) identificar métodos longos de acordo com os valores encontrados na tarefa anterior. A saída desse processo é uma lista contendo todos os métodos do sistema em desenvolvimento.

Os interesses arquiteturais são definidos com base em duas informações: (1) o papel arquitetural da classe e (2) o design do código da aplicação. Em sistemas que seguem arquiteturas referências [Medvidovic & Taylor 2010], o papel arquitetural da classe é normalmente atribuído através de herança, anotação ou implementação de uma

interface provida por essa arquitetura referência. Por exemplo, em aplicações que seguem a arquitetura referência ASP.NET MVC[1], uma classe possui o papel de *Controller* quando estende uma classe abstrata com o mesmo nome. Já aplicações que seguem a arquitetura referência Spring WEB MVC[2], essa atribuição é realizada através da inserção da anotação *@Controller*. Entretanto, decisões de design podem atribuir responsabilidades adicionais a um papel arquitetural. Por esse motivo a técnica proposta também considera essas decisões de design extraídas do código do sistema referência.

2.1. Exemplo de Uso

Para exemplificar a utilização da ferramenta utilizaremos duas versões do sistema MobileMedia [Figueiredo et al. 2008]. A primeira versão foi utilizada como sistema referência de *design* e a segunda versão foi utilizada como o sistema que precisa ser avaliado. A utilização de versões próximas tem como objetivo usar sistemas com poucas diferenças em relação às decisões *design*.

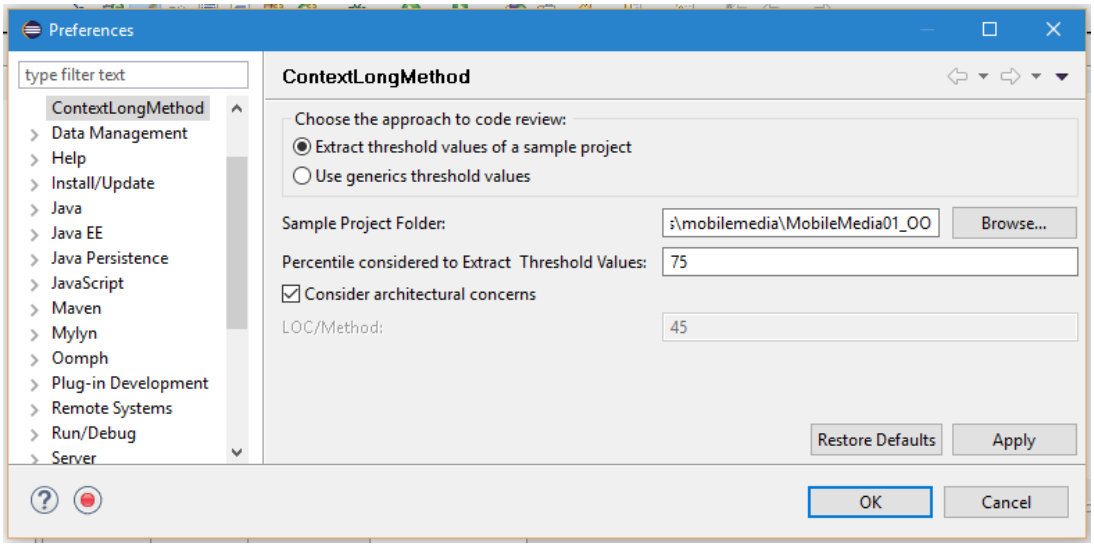


Figura 3. Tela de preferências do *plug-in*.

A utilização do *plug-in* inicia-se com a seleção de uma das três abordagens disponíveis para identificação de métodos longos. No ambiente Eclipse essa opção é acessada através do menu **Window** > **Preferences**. A Figura 3 apresenta a tela de preferências na qual é possível selecionar um dos três mecanismos para identificação de métodos longos detalhados na Seção 2. Essas opções foram criadas para facilitar a comparação dos resultados obtidos com a ferramenta. A Figura 3 exemplifica as configurações para utilizar os interesses arquiteturais para identificação de métodos longos. O valor limiar padrão para a métrica SLOC/Método é o percentil 75, ou seja, o limiar máximo da métrica engloba 75% dos métodos do interesse arquitetural.

Para habilitar a realização de análises deve-se pressionar o botão direito sobre cada sistema e selecionar a opção *Enable ContextLongMethod Analysis.* Ao selecionar essa opção já será realizada a primeira análise em todas as classes do sistema. Essa análise é realizada em paralelo à codificação e pode demorar alguns segundos caso exista um grande número de classes no sistema. Entretanto, as próximas análises só irão considerar as classes cujas modificações foram salvas pelo desenvolvedor e ocorrem

[1] http://www.asp.net/mvc
[2] http://projects.spring.io/spring-framework/

rapidamente. Para interromper a execução das análises no sistema deve-se selecionar a opção *Disable ContextLongMethod Analysis.*

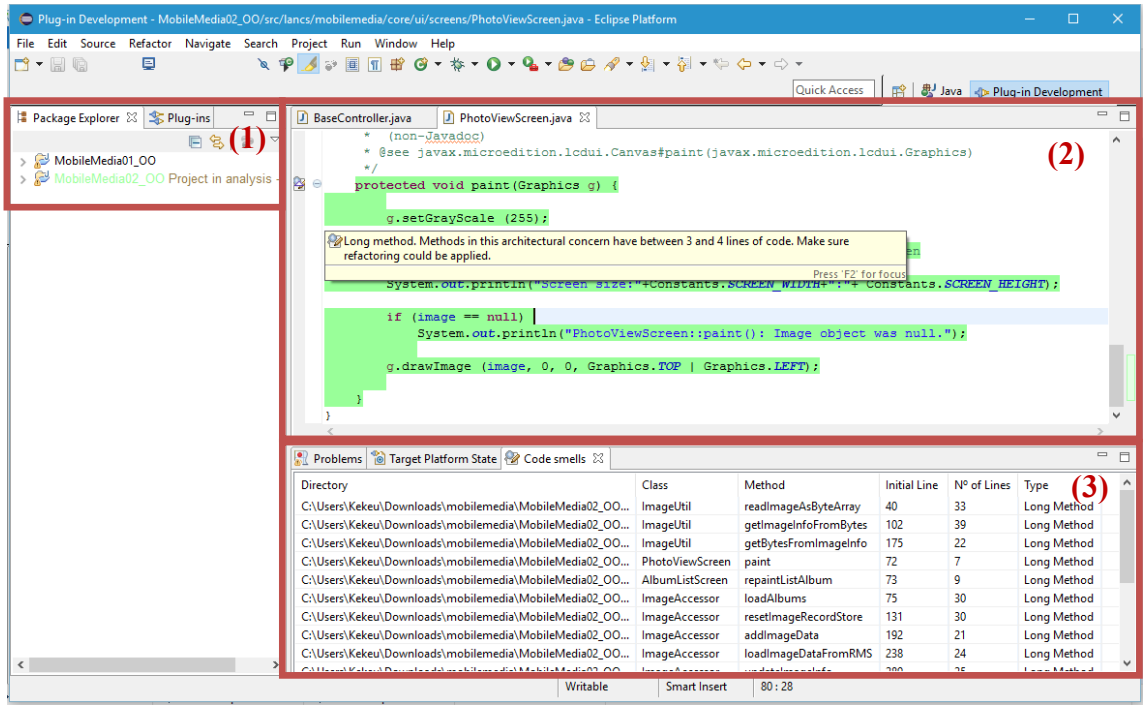


Figura 4. Informações sobre o Método Longo Identificado pelo *plug-in*.

A Figura 4 mostra os três mecanismos disponibilizados pela ferramenta *ContextLongMethod* para recomendar métodos longos para o desenvolvedor. O mecanismo (1) identifica no *Package Explorer* do Eclipse os sistemas que estão sendo analisados pela ferramenta. O mecanismo (2) mostra ao desenvolvedor informações detalhadas sobre os métodos longos identificados na classe ativa. O código fonte de cada método é pintado na classe. Uma sinalização a direita do editor de texto permite navegar mais rapidamente entre todos os métodos longos identificados na classe ativa. Outra marcação ao lado esquerdo do editor exibe uma mensagem informando que, no interesse arquitetural analisado, os métodos possuem entre 3 e 4 linhas. O método analisado possui 7 linhas e foi considerado longo pela técnica. Nesse método a existência de comandos de *logging,* provavelmente desnecessários, foram os responsáveis por incrementar o tamanho do método em relação aos métodos pertencentes ao mesmo interesse arquitetural. Finalmente o mecanismo (3) é a *View Code Smells*, que exibe uma lista com todos os métodos identificados como longos no sistema em análise.

2.2. Arquitetura da Ferramenta

A arquitetura do *plug-in* utiliza apenas bibliotecas disponibilizadas pelo próprio Eclipse. A Figura 5 exibe os principais componentes dessa arquitetura. O componente *threshold provider* contém as informações sobre os valores limiares obtidos do projeto referências. Esses valores são utilizados de acordo com as opções selecionadas no componente *Preferences.* Quando o componente *enable Analysis* é ativado no Eclipse as análises no código são iniciadas de acordo com os valores limiares armazenados e as preferências selecionadas.

A última etapa atualiza as *Views* para apresentar informações dos métodos longos identificados de acordo com a abordagem seleionada.

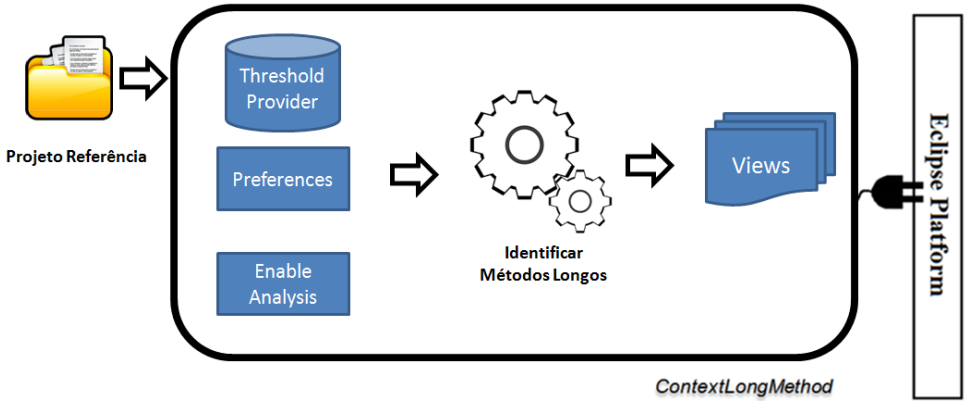


Figura 5. Principais componentes da arquitetura de *ContextLongMethod*.

2.3. Avaliação

Para avaliar os resultados obtidos com a ferramenta *ContextLongMethod* utilizamos nove versões do sistema MobileMedia, uma linha de produtos de *software* para aplicações que manipulam foto, música e vídeo em dispositivos móveis [Figueiredo et al. 2008]. Usamos como base o estudo realizado por Paiva et al. (2015), que analisou a acurácia das ferramentas inFusion, JDeodorant e PMD para detectar três *code smells: feature envy, god class e god method*. Para isso os autores usaram uma lista de referência construída por especialistas com os *code smells* encontrados por eles no código fonte das versões do MobileMedia. Usamos a mesma lista de referência de *god methods* para calcular a precisão, a cobertura (*recall*) e F-Score da ferramenta *ContextLongMethod*. A métrica F-Score pondera igualmente os valores de precisão e cobertura, sendo uma medida utilizada para quantificar a acurácia.

Consideramos fazer sentido comparar os resultados da detecção de métodos longos com a detecção de *god method*, pois, de acordo com sua definição, um *god method* é um método que cresceu muito e que concentra a maior parte da funcionalidade da sua classe. A Tabela 1 exibe os resultados sumarizados do estudo de Paiva et al. (2015) e os obtidos com a ferramenta *ContextLongMethod* utilizando cinco possibilidades de configuração. Maiores detalhes sobre os valores obtidos na avaliação estão disponíveis no site[3].

Percebe-se que todos os valores do F-Score foram superiores na ferramenta proposta. Na primeira configuração, a *ContextLongMethod* foi usada com um valor limiar fixo de 45 LOC/Método, escolhido por conveniência. Apesar do bom resultado obtido, normalmente é muito difícil chegar a um valor limiar fixo que seja interessante para todo sistema. Também avaliamos duas configurações usando a primeira versão do sistema MobileMedia (V1) como referência e os percentis 75 e 90. Os dois valores do F-Score foram superiores ao obtido nas ferramentas existentes. Finalmente as duas últimas linhas da tabela apresentam os resultados quando consideramos os percentis 75 e 90 e também os interesses arquiteturais. A primeira versão (V1) do MobileMedia também foi usada como sistema referência nessas análises. Nestes dois casos, as pontuações do F-Score foram superiores aos valores encontrados quando o contexto não foi considerado. O valor de precisão e cobertura exibidos nas linhas da tabela correspondem ao resultado das análises nas nove versões do MobileMedia.

[3] https://sites.google.com/site/cbsoft2016/

Tabela 1. Resultados da ContextLongMethod e de Paiva et al. (2015)

		Precisão (%)	Cobertura (%)	% F-Score
Paiva et al. (2015)	inFusion	100	26	41,27
	Jdeodorant	35	50	41,18
	PMD	100	26	41,27
ContextLongMethod	45 LOC/Método	96	47	63,10
	Percentil 75	27	100	42,52
	Percentil 90	56	95	70,46
	Percentil 75 + interesse	32	100	48,48
	Percentil 90 + interesse	60	89	71,68

Vale ressaltar que quando consideramos o percentil 75 a ferramenta *ContextLongMethod* obteve 100% de *recall*, ou seja, todos os métodos indicados pelos especialistas foram identificados. Apesar de gerar ainda alguns falsos positivos, a ferramenta não gerou nenhum falso negativo que são mais difíceis de serem identificados no código.

3. Ferramentas Relacionadas

JDeodorant[4] é um *plug-in open source* para o Eclipse que detecta quatro *code smells* entre eles *God Methods*. O desenvolvedor precisa executar manualmente as análises e configurar valores limiares genéricos que são usados para todas as classes. A ferramenta que propusemos também é um *plug-in open source* para o Eclipse, mas detecta os métodos longos em tempo real. Os valores limiares são detectados automaticamente a partir do interesse arquitetural da classe.

PMD[5] é uma ferramenta *standalone* e também possui um *plug-in* para integração com o Eclipse que detecta vários *code smells*, entre eles os métodos longos. Utiliza também apenas a métrica de número de linhas de código e um valor limiar genérico que pode ser configurado, mas não considera o interesse arquitetural da classe analisada. No Eclipse, também é necessário que o desenvolvedor execute manualmente as análises.

SonarLint[6] é um *plug-in* para o eclipse que fornece para os desenvolvedores *feedback* em tempo real sobre novos bugs e problemas de qualidade que são inseridos em projetos Java, JavaScript e PHP. A análise do SonarLint é acionada quando o arquivo é aberto ou salvo e também utiliza valores limiares genéricos que podem ser configurados. Apesar de também fazer a avaliação em tempo real como a nossa ferramenta, o SonarLint não considera os interesses arquiteturais do sistema avaliado para calibrar os valores limiares que devem ser ajustados manualmente.

4. Considerações Finais

Neste artigo apresentamos a ferramenta *ContextLongMethod,* uma ferramenta que extrai informações contextuais do *design* de um sistema referência e utiliza-as para avaliar um código em desenvolvimento, recomendando em tempo real possíveis problemas para o

[4] Disponível no site http://www.jdeodorant.org/
[5] Disponível no site http://pmd.github.io/
[6] Disponível no site http://www.sonarlint.org/eclipse/

desenvolvedor. Na avaliação inicial da ferramenta, a abordagem proposta obteve melhores resultados em relação às abordagens existentes, em alguns casos conseguindo recuperar todos os métodos longos identificados por especialistas.

Como trabalhos futuros, estamos estendendo a utilização das informações contextuais para detecção de outros *code smells*. Também estamos aprimorando o algoritmo que recupera informações do contexto arquitetural do sistema referência e pretendemos fazer avaliações em outros sistemas maiores para melhorar o nível de evidência dos resultados. *ContextLongMethod* é um plug-in sob licença *open source* e disponível no site https://github.com/marcosdosea/ContextSmellDetector

Agradecimentos. Esse trabalho foi apoiado pelo CNPq: Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (processo 573964/2008–4) e Projeto Universal (processo 486662/20136)

References

Arcoverde, R. et al., 2012. Automatically detecting architecturally-relevant code anomalies. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, pp. 90–91.

Dósea, M., Sant'Anna, C.N. & Santos, C., 2016. Towards an Approach to Prevent Long Methods Based on Architecture-Sensitive Recommendations. In *Forth Workshop on Software Visualization, Evolution and Maintenance (VEM 2016)*.

Figueiredo, E. et al., 2008. Evolving software product lines with aspects. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA, New York, USA: ACM Press, p. 261.

Fontana, F.A. et al., 2013. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *2013 IEEE International Conference on Software Maintenance*.

van Gurp, J. & Bosch, J., 2002. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2), pp.105–119.

Marinescu, R., 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance*.

Medvidovic, N. & Taylor, R.N., 2010. Software architecture. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. New York, New York, USA: ACM Press, p. 471.

Paiva, T. et al., 2015. Experimental Evaluation of Code Smell Detection Tools. *3th Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015)*.

Palomba, F. et al., 2013. Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp.268–278.

Robillard, M.P., Walker, R.J. & Zimmermann, T., 2010. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4), pp.80–86.

Zhang, F. et al., 2013. How Does Context Affect the Distribution of Software Maintainability Metrics? In *2013 IEEE International Conference on Software Maintenance*. IEEE, pp. 350–359.

Codivision: Uma Ferramenta para Mapear a Divisão do Conhecimento entre os Desenvolvedores a partir da Análise de Repositório de Código

Francisco Vanderson de Moura Alves[1], **Werney Ayala Luz Lira**[1], **Irvayne Matheus de Sousa Ibiapina**[1], **Pedro de Alcântara dos Santos Neto**[1]

[1]Easii - Laboratório de Engenharia de Software e Informática Industrial
DC - Departamento de Computação
CCN - Centro de Ciências da Natureza
UFPI - Universidade Federal do Piauí
Brasil

vanderson.moura.vm@gmail.com, werney.zero@gmail.com

irvaynematheus@yahoo.com, pasn@ufpi.edu.br

Abstract. *Software development by a team allows the construction of solutions for major problems and shorter delivery time. However, an individual of a team canconcentrate on all the contributions for a software component, creating a bad scenario for teamwork. This work presents the Codivision tool, that allows viewing the level of contribution of developers to a project, warning when such levels pose risks to continuity.*

Resumo. *O desenvolvimento de software em equipe permite a construção de soluções para problemas maiores e com um menor tempo de entrega. Porém, o trabalho em equipe pode introduzir o domínio sobre o conhecimento de certas partes do produto por apenas um membro da equipe. Neste trabalho é apresentada a ferramenta Codivision, que permite visualizar o nível de contribuição de desenvolvedores a um projeto, alertando quando tais níveis representem riscos para sua continuidade.*

Vídeo disponível em: `https://youtu.be/eRvN_1e6vKY`

1. Introdução

Um processo de desenvolvimento de software consiste basicamente em um conjunto de atividades organizadas com o objetivo de se construir um produto [Koscianski and Soares 2007]. A qualidade de um produto de software é fortemente dependente da qualidade do processo pelo qual ele é construído e mantido [Rocha et al. 2001]. Com isso, é muito importante que cada uma das etapas do processo de desenvolvimento seja executada da melhor forma possível.

Durante a implementação é comum que a equipe participante seja dividida em grupos, de tal forma que cada grupo se responsabilize por uma parte específica do produto. Porém, essa divisão do trabalho pode levar ao "domínio" de parte do software, ou seja, do código associado a essa parte, por apenas um grupo de pessoas, ou em um nível mais extremo, por apenas uma única pessoa. Desse fato decorre, por exemplo, a dificuldade de

manutenção do software, sendo exigido que tal pessoa participe de toda e qualquer ação que envolva a parte do software de sua "propriedade" [Teles 2004]. Outro complicador associado a esse fato, é o comprometimento da qualidade e legibilidade do código, uma vez que existe, fundamentalmente, apenas uma opinião sobre uma área específica de um projeto.

As metodologias ágeis já se preocuparam de forma explícita com essa questão, tanto que criaram uma prescrição que exige que o código seja coletivo, incentivando assim que várias pessoas atuem nas mais variadas áreas, ao mesmo tempo em que incentiva o trabalho em pares, permitindo assim que mais de uma pessoa esteja associada a todo e qualquer código desenvolvido. Fora isso, há a necessidade de uma maior coletividade de código quando se trata do fato de prescrever o desenvolvimento guiado por testes e refatoração contínua, com isso há uma contribuição no processo de evitar o "domínio" de partes do software por um desenvolvedor ou por um pequeno grupo de desenvolvedores [Beck et al. 2001].

Nesse contexto, foi desenvolvida uma ferramenta, intitulada *Codivision*, que se baseia em técnicas de Mineração de Repositórios de Software [Hassan 2008], para apoiar equipes na descoberta da distribuição de conhecimento dos desenvolvedores sobre o código-fonte de um projeto de software. Tal conhecimento é estimado a partir do nível de contribuição de um indivíduo ao projeto. A partir do uso dessa ferramenta, gerentes de projetos podem identificar de forma precisa que partes do software são "dominadas" por pequenos grupos de desenvolvedores, e assim gerar estratégias que possam contornar esse fato, reduzindo assim os problemas que podem ser ocasionados, afim de melhorar o processo de desenvolvimento de software e consequentemente a qualidade do produto gerado.

Este trabalho apresenta a ferramenta *Codivision* e está organizado da seguinte forma: a Seção 2 apresenta alguns trabalhos relacionados; as Seções 3 e 4 apresentam detalhes da arquitetura e funcionalidades da ferramenta, respectivamente. A Seção 5 apresenta um exemplo de uso da ferramenta. Por fim, a Seção 6 apresenta algumas conclusões do trabalho.

2. Trabalhos Relacionados

A estimação do conhecimento de desenvolvedores sobre sistemas de software pode ser feito pela análise das operações realizadas sobre o código fonte. Ferramentas que automatizam todo o processo de análise e apresentação de resultados, tal como a apresentada neste trabalho, ainda não puderam ser identificadas. Apesar disso, existem alguns estudos que apresentam métodos e técnicas de mineração de repositórios de software para criação de métricas que auxiliam na avaliação das atividades realizadas por desenvolvedores, como por exemplo, o número de aquivos criados ou modificados.

Grande parte dos estudos que visam estimar o conhecimento de desenvolvedores sobre o código-fonte, utilizam técnicas de mineração de dados de Sistemas de Controle de Versão (SCV) [Otte 2009], tais como [Fritz et al. 2014], [Moura et al. 2014] e [Yu and Ramaswamy 2007]. O primeiro modela o conhecimento de desenvolvedores por meio das alterações de código em nível de arquivos. O segundo, apresenta métricas de menor granularidade (alterações em nível de linhas) em relação as operações realizadas pelos desenvolvedores sobre o código-fonte. O terceiro, também leva em consideração

métricas que representam alterações em nível de linhas de código, mas utilizam especificamente sistemas de versionamento CSV.

A ferramenta *Codivision* realiza mineração de dados dos principais tipos de sistemas de versionamento (Git ou SVN). Além de considerar alterações em nível de arquivo e linhas de código na estimação do conhecimento de desenvolvedores, é considerado também o que acontece em um contexto real, e o cálculo das alterações de código simula a degradação do conhecimento dos desenvolvedores, ou seja, é representada a "perda" de conhecimento por período de tempo transcorrido e também por novas alterações realizadas por outros desenvolvedores nas mesmas entidades de código.

3. *Codivision*: Visão Geral

A ferramenta proposta neste trabalho possui dois componentes principais. O primeiro (composto pelos módulos em cor azul) é responsável pela visualização de informações de um repositório, desenvolvido seguindo o modelo MVC (Model View Controller), e que apresenta métricas calculadas com base nas contribuições dos desenvolvedores a um projeto. O segundo componente (composto pelos módulos em cor verde) é responsável pela conexão com os repositórios de código e extração de dados para o cálculo de métricas. Um diagrama exibindo a arquitetura da ferramenta pode ser visto na Figura 1.

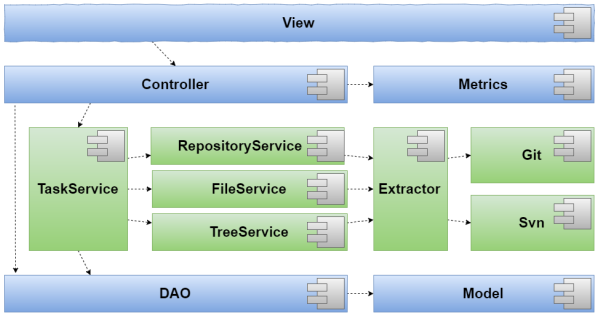


Figura 1. Arquitetura da ferramenta

A arquitetura foi pensada de forma que a extração de dados funcionasse independente da interação com o usuário, executando de forma paralela. Isso foi necessário porque a extração depende de alguns fatores externos, como a conexão de rede a um servidor com o repositório de código e esse processo demanda algum tempo.

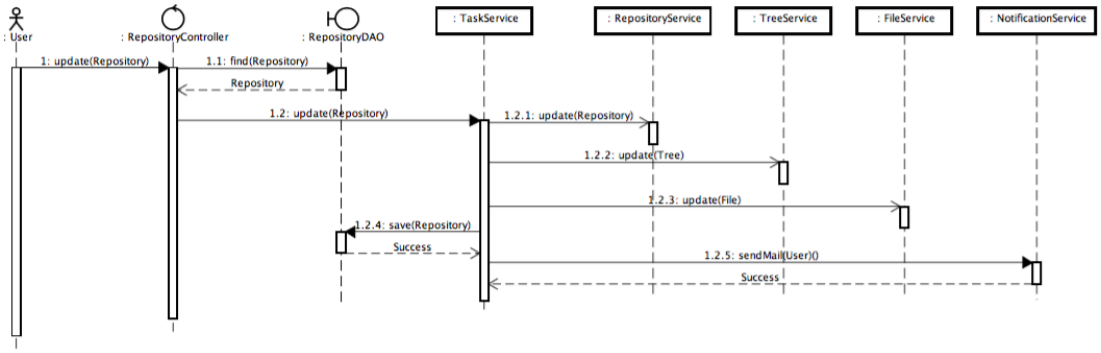


Figura 2. O processo de extração de dados de um repositório de código.

A extração utiliza um *pool* de *threads* baseado na quantidade de processadores disponíveis. Para realizar a extração de um repositório são necessárias várias *threads* (serviços). O primeiro serviço (RepositoryService) é responsável por extrair as informações das revisões (*commits*) feitos no repositório. Enquanto isso, um outro serviço (TreeService) extrai a árvore de diretórios do projeto. Por fim, são instanciados vários serviços (FileService) para extrair as informações de cada arquivo enviado em cada *commit*. Esse processo de atualização ou extração de um repositório pode ser visto na Figura 2.

4. Principais Funcionalidades

Nesta seção serão descritas as principais funcionalidades da ferramenta. Como a interação com o usuário é algo relativamente simples, ela será explanada apenas na Seção 5, que mostra um exemplo de uso, assim como a funcionalidade de visualização dos dados.

4.1. Funcionalidade 1: Extração dos Dados

Para se iniciar o uso da *Codivision*, é necessário configurar um repositório para ter seus dados extraídos. Esse passo consiste em identificar os responsáveis por cada *commit* feito no repositório. A partir dos *commits*, pode-se obter os arquivos que foram modificados e assim determinar quais linhas foram alteradas. Com isso, é possível mensurar a quantidade de contribuições feitas por cada desenvolvedor no projeto.

As alterações podem ser analisadas de duas maneiras. O primeiro tipo de análise é feito com base em um arquivo alterado como um todo, por exemplo: ao modificar qualquer parte de um arquivo, isso representará uma alteração, mesmo que tenha sido alterado 1 (uma) ou várias linhas de código. A segunda maneira, consiste em identificar cada linha alterada em um arquivo. Essas alterações podem ser de três tipos: adição, deleção ou modificação.

Considera-se adição a inclusão de uma nova linha no arquivo, bem como a deleção é uma remoção de linha. A modificação pode ser considerada de duas maneiras: uma delas é considerar qualquer adição ou deleção como uma modificação, o que resultaria na soma desses dois indicadores; a outra maneira, que é a forma utilizada neste trabalho, utiliza o algoritmo de Levenshtein [Sankoff and Kruskal 1983], que calcula a diferença entre duas *Strings*, que é dada pelo número de operações necessárias para transformar uma *String* na outra. Desse modo, verifica-se se em um conjunto adição/deleção se o valor dado pelo algoritmo de Levenshtein é menor que 75% do tamanho dessa *String*, em caso afirmativo, não houve uma adição seguido de deleção, mas uma modificação.

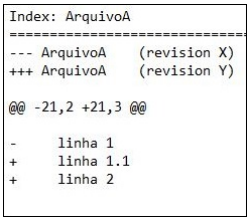


Figura 3. Exemplo de diff

As linhas alteradas em cada arquivo dos *commits* podem ser obtidas por meio de um arquivo de *diff*. A operação *diff* exibe exatamente o que mudou nesse arquivo entre

uma versão e outra. A Figura 3 mostra um exemplo de *diff* no formato unificado. As duas primeiras linhas ("—" e "+++") indicam o arquivo no qual está sendo feito o *diff* e a versão desse arquivo. Em seguida, podem ocorrer um ou mais trechos que iniciam com "@@‘’, que apresentam a linha inicial do trecho que segue e a quantidade de linhas desse trecho na versão anterior ("-") e na versão atual ("+") do arquivo respectivamente. Em seguida, são exibidas as linhas adicionadas na versão atual ("+") e linhas que existiam na versão anterior, mas foram removidas na versão atual ("-") e as linhas inalteradas.

Além das alterações citadas no parágrafo anterior, é extraída a quantidade de linhas alteradas que envolvem um comando condicional (*if*). Na *Codivision* existe a opção de conceder maior peso às alterações realizadas sobre linhas de código que contenham comandos condicionais.

Durante a extração dos dados não é armazenado o código dos projetos extraídos. São armazenadas apenas informações referentes às mudanças realizadas. Os *diffs* obtidos são utilizados apenas para extrair a quantidade de alterações (adições, modificações e deleções) realizadas em cada *commit*.

4.2. Funcionalidade 2: Refinamento dos Dados

A ferramenta *Codivision* gera um valor para as contribuições baseada nos tipos de alteração identificados nos diversos *commits* realizados ao projeto. Nesta seção serão brevemente descritos os fatores usados para esse refinamento.

4.2.1. Tipo de Alteração

Como apresentado anteriormente, são considerados 3 tipos de alteração (adição, modificação e deleção), porém, com um ponderador adicional para alterações realizadas em linhas contendo comandos condicionais (que envolvem *ifs*), pois geralmente representam as regras de negócio de um sistema, e assim demandam um maior conhecimento por parte dos desenvolvedores durante a alteração. É importante notar que cada tipo de alteração possui uma esforço diferente para realizá-la. Por exemplo, remover uma linha é mais simples do que adicionar uma nova linha. Desse modo, percebeu-se a necessidade de atribuir pesos a cada tipo de alteração, com a finalidade de balancear o nível de contribuição ao projeto associado a cada tipo de alteração. A *Codivision* define como padrão os seguintes valores de ponderação: adição (1,0), deleção (0,5), modificação (1,0) e comando condicional (1,0). Estes valores foram definidos de forma empírica. Contudo, a ferramenta *Codivision* permite que o usuário realize alterações de acordo com sua percepção.

4.2.2. Degradação por Tempo

Algo bastante comum do ser humano é ter parte do conhecimento adquirido esquecido com o passar do tempo. Quando se trata de código isso não é diferente. Um desenvolvedor que fique por muito tempo sem alterar um código tem seu conhecimento degradado, exigindo um esforço maior para uma manutenção nessa parte do projeto.

A métrica de degradação do conhecimento por tempo visa simular o esquecimento como algo natural do ser humano. Essa degradação é calculada da seguinte maneira:

um pequeno valor, que aumenta de acordo com a quantidade de dias passados desde a data do *commit*, é subtraído da quantidade total de contribuições ao projeto feitas pelo desenvolvedor.

4.2.3. Degradação por Nova Alteração

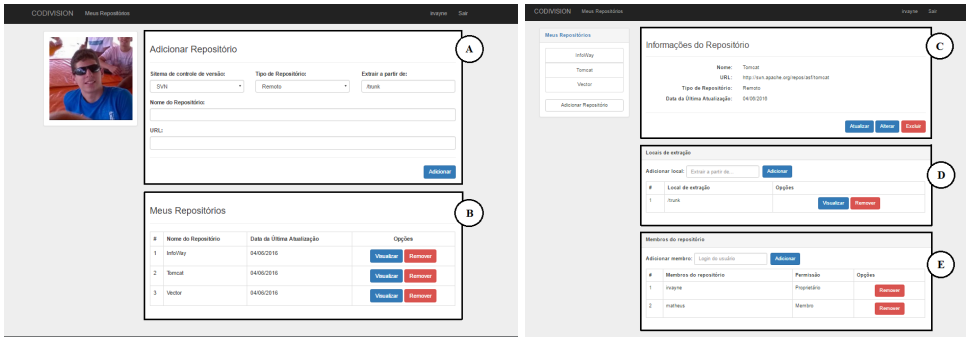
O conhecimento que um membro da equipe possui sobre um arquivo pode ser determinado pela quantidade de alterações feitas por ele. Mas é importante observar que como o projeto é desenvolvido em equipe, vários desenvolvedores podem alterar os mesmos arquivos constantemente. Como consequência disso, as alterações feitas por um membro podem ser sobrescritas por outro, ou um desenvolvedor diferente pode adicionar um novo conteúdo e com isso o conhecimento inicial acerca do arquivo pode não ser o mesmo após essas várias alterações terem sido feitas sobre o arquivo.

A métrica de degradação por nova alteração pretende simular o impacto de novas alterações, feitas por outros membros da equipe, no conhecimento acerca desse arquivo, para o membro que fez as alterações anteriores. Para isso, um pequeno valor é subtraído da quantidade total de alterações, baseado na quantidade de alterações realizadas por outros desenvolvedores, após a versão do arquivo em que está sendo feito o cálculo.

5. Utilizando a Ferramenta *Codivision*

A *Codivision* é uma ferramenta desenvolvida para plataforma Web. Ela é uma ferramenta para visualização da contribuição de cada desenvolvedor em um projeto de software. Todas as funcionalidades podem ser acessadas após um cadastro prévio do usuário, feito na própria ferramenta. O acesso à *Codivision* pode ser feito por meio da seguinte URL: http://easii.ufpi.br/codivision/.

A Figura 4(a) apresenta a página inicial visível ao usuário após o *login*. Nessa página é possível adicionar um novo repositório e listar todos os repositórios já cadastrados. Para adicionar um novo repositório é necessário preencher os campos do formulário que se encontram na área destacada A. A área em destaque B apresenta a lista de repositórios já adicionados.



(a) Página inicial do usuário. (b) Página de informações do repositório.

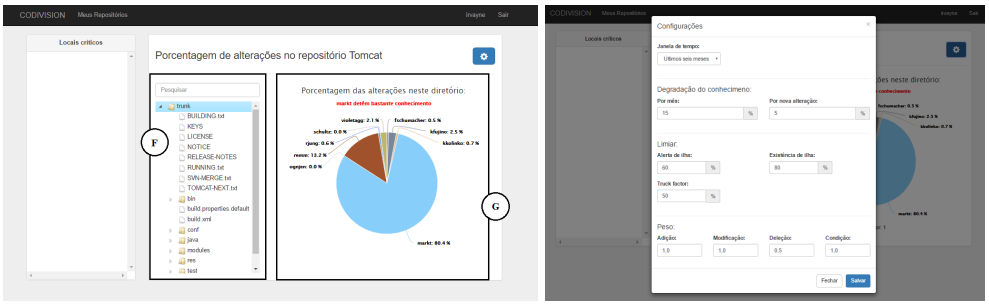
Figura 4. Adicionando e listando repositórios.

É possível visualizar informações detalhadas sobre os repositório. Para isso, deve ser escolhido a opção visualizar que se encontra na área destacada B da Figura 4(a). As

informações sobre o repositório são apresentadas na Figura 4(b), na área em destaque C. Nessa mesma área é possível realizar algumas operações no repositório como atualizar, editar e excluir.

Na área destacada D, apresentada na Figura 4(b), podem ser visualizados todos os locais de extração. Nela é possível adicionar ou excluir locais de extração e visualizar a árvore de diretório que representa esses locais. Já a área destaca E apresenta a lista de todos os membros desse repositório em questão. Dependendo da permissão do usuário, é possível adicionar ou excluir membros ao repositório.

Quando é escolhido a opção visualizar a árvore de diretório de um local específico, o usuário é direcionado para a página ilustrada na Figura 5(a). Nela é possível ver as pastas e arquivos que compõem esse local. O que está destacado em F apresenta exatamente isso. O que está destacado em G é o gráfico onde mostra o nível de contribuição de cada desenvolvedor, de acordo com o módulo ou arquivo selecionado.



(a) Gráfico do repositório. (b) Configurações.

Figura 5. Visualizando as alterações em cada parte do projeto.

É possível modificar alguns parâmetros utilizados no cálculo da métrica principal que indica o nível de contribuição de um desenvolvedor ao projeto. Isso pode ser feito por meio da opção de configurações, que é representada por uma engrenagem azul que se encontra na região superior à direita da Figura 5(a).

A Figura 5(b) mostra a janela de configurações com as opções que podem ser alteradas. Pode ser definido qual o período de tempo em que devem ser contabilizados o percentual de alterações (essa opção permite, por exemplo, obter as alterações realizadas no início ou durante os últimos meses do projeto), o percentual de degradação do conhecimento por mês e por nova alteração, o limiar para alerta de partes do código "dominadas" por um determinado desenvolvedor, além das opções de definição de pesos para operações de adição, modificação, deleção de linhas de código. Além disso, as operações realizadas em comandos condicionais podem ser priorizadas com um peso maior.

6. Conclusão

Este trabalho apresentou uma ferramenta que possibilita aos seus usuários a obtenção de uma grande quantidade de informações relacionadas tanto à equipe de desenvolvimento quanto ao projeto. Ela pode ser utilizada tanto em repositórios Git quanto SVN. Por meio da ferramenta pode-se obter também informações a partir de repositórios públicos ou privados, sejam remotos ou locais (somente acessíveis a partir de uma intranet).

A *Codivision* foi criada com o intuito de minimizar os riscos de fracasso do projeto. Desse modo, o gerente de projeto pode utilizá-la para distribuir melhor as tarefas da equipe de desenvolvimento, de modo que o conhecimento sobre o código seja compartilhado entre todos os desenvolvedores. Além disso, a equipe pode utilizá-la para descobrir o detentor de maior conhecimento para ajudá-la na realização de tarefas.

A ferramenta já foi utilizada na avaliação de vários repositórios públicos e privados, tanto do Git quanto do SVN e os resultados mostraram-se bastante promissores. A partir dessa avaliação foi possível demonstrar os pontos positivos do uso da ferramenta e algumas empresas já manifestaram interesse em utilizá-la durante o processo de desenvolvimento. Como proposta futura, pretende-se realizar estudos de caso nessas empresas, a fim de avaliar o impacto do uso da ferramenta em projetos de software.

7. Agradecimentos

Agradecemos à empresa de desenvolvimento de software *Infoway*, por fornecer apoio na realização de testes experimentais da ferramenta desenvolvida por meio da extração de dados de projetos criados pela própria empresa.

Referências

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). Manifesto for agile software development.

Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):14.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE.

Koscianski, A. and Soares, M. d. S. (2007). Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software.

Moura, M. H. D. d., Nascimento, H. A. D. d., and Rosa, T. C. (2014). Extracting new metrics from version control system for the comparison of software developers. In *Software Engineering (SBES), 2014 Brazilian Symposium on*, pages 41–50. IEEE.

Otte, S. (2009). Version control systems. *Computer Systems and Telematics, Institute of Computer Science, Freie Universität, Berlin, Germany*.

Rocha, A. R. C. d., Maldonado, J. C., and Weber, K. C. (2001). *Qualidade de software*. São Paulo: Prentice Hall.

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley.

Teles, V. M. (2004). Extreme programming. *São Paulo: Novatec*.

Yu, L. and Ramaswamy, S. (2007). Mining cvs repositories to understand open-source project developer roles. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 8–, Washington, DC, USA. IEEE Computer Society.

Cronos IDE: Uma Ferramenta Web para o Desenvolvimento de Aplicações Java na Nuvem

Flávio R. C. Sousa[2], **Maristella Ribas** [1], **Lincoln S. Rocha** [3]

[1] Techne Engenharia e Sistemas
Av. das Nações Unidas, 10989 - Chácara Itaim, SP – Brasil

[2]Departamento de Engenharia de Teleinformática
Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brasil

[3]Departamento de Computação
Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brasil

sousa@ufc.br, mari@techne.com.br, lincoln@dc.ufc.br

Abstract. *Cloud Computing provides on demand services in a pay-per-use basis. In this environment, users can access services at any time and from any place. The quantity and variety of services, such as processing and storage have increased. In this context, a particularly important type of service is an IDE for software development in the cloud. However, available cloud IDEs either do not provide features for rapid development, or the developed software is tied to a particular cloud platform for execution. This paper presents Cronos IDE, a web tool for cloud applications development. Cronos IDE uses data modeling to create the back-end and front-end of the application in a fast and simple way, and presents features to create easy integration to other services.*

Resumo. *Computação em nuvem tem como objetivo proporcionar serviços sob demanda com pagamento baseado no uso. Neste ambiente, os usuários podem acessar os serviços a qualquer instante e de qualquer local, o que tem aumentado a quantidade de serviços disponíveis, tais como armazenamento e processamento. Neste contexto, destaca-se o uso de IDEs para o desenvolvimento de software em nuvem. Entretanto, as IDEs existentes não permitem o desenvolvimento rápido ou as aplicações desenvolvidas são dependentes de plataforma de nuvem para sua execução. Este artigo apresenta a Cronos IDE, uma ferramenta web para o desenvolvimento de aplicações na nuvem. Cronos IDE utiliza a modelagem dos dados para a criação do* back-end *e* front-end *da aplicação de forma rápida e simples, além de facilitar a integração com outros serviços.*

Vídeo: `https://youtu.be/qRMZyffUNHQ`

1. Introdução

Computação em nuvem é uma tendência de tecnologia cujo objetivo é proporcionar serviços sob demanda com pagamento baseado no uso. Neste ambiente, as empresas podem alugar capacidade de processamento e armazenamento, reduzindo seus custos. A computação em nuvem surge da necessidade de construir infraestruturas de TI complexas, onde os usuários têm que realizar instalação, configuração e atualização de sistemas de

software. Além disso, recursos de computação e hardware são propensos a ficarem obsoletos rapidamente. Assim, a utilização de plataformas computacionais de terceiros é uma solução inteligente para os usuários lidarem com infraestrutura de TI. Na computação em nuvem os recursos de TI são fornecidos como um serviço, permitindo que os usuários o acessem sem a necessidade de conhecimento sobre a tecnologia utilizada. Assim, usuários e empresas passaram a acessar os serviços sob demanda e independente de localização, o que aumentou a quantidade de serviços disponíveis [Sousa and Machado 2014].

Existe uma crescente utilização de ambientes de computação de forma global, nacional e regional, tanto por empresas e indústrias quanto pelo segmento governamental. O paradigma de nuvem, que era vista com desconfiança no Brasil, avançou em 2013 e o ritmo de crescimento tende a aumentar continuamente. De acordo com o Gartner, para 2017 os negócios nessa área irão se multiplicar no país e gerar uma receita de aproximadamente $ 4,5 bilhões de dólares.

A construção de software para a nuvem é um grande desafio na consolidação deste tipo de software [Aho et al. 2011]. Com isso, é fundamental o uso de IDEs e ferramentas específicas para o desenvolvimento de software para o ambiente de nuvem. As IDEs em nuvem devem permitir a construção de aplicações que utilizem um modelo de programação flexível e que utilizem as principais características dos ambiente de computação em nuvem, tais como serviço sob demanda, elasticidade e serviço medido. Deve-se evitar o aprisionamento do cliente, que é um grande inibidor da adoção dos modelos de nuvem em geral. Assim, seguindo esse direcionamento, será possível melhorar a adoção de aplicações em nuvem em conjunto aos diversos seguimento de mercado.

A maioria dos fornecedores de plataforma como serviço (PaaS) não oferecem uma ferramenta com interface gráfica que melhore a produtividade no desenvolvimento de aplicação para o ambiente de computação em nuvem [Mutiara et al. 2014]. Existem algumas IDEs para o desenvolvimento de aplicações na nuvem, entretanto as IDEs existentes não permitem o desenvolvimento rápido ou as aplicações desenvolvidas são dependentes de plataformas de nuvem específicas para executarem. Além disso, estas IDEs não contemplam as características essenciais da nuvem (e.g., elasticidade e serviço medido). De acordo com nosso estudo, as IDEs existentes abordam de forma incipiente e parcial essas limitações tornando necessário o desenvolvimento de novas ferramentas.

Com o objetivo de atacar esse problema, este trabalho propõe a Cronos IDE, uma ferramenta web para o desenvolvimento de aplicações Java na nuvem. Com a Cronos IDE, os desenvolvedores não precisam de software pré-instalados para iniciar a colaboração e o desenvolvimento de aplicações. Para melhorar o desenvolvimento, a Cronos IDE utiliza a modelagem dos dados para a criação do *back-end* e *front-end* da aplicação de forma rápida e simples, além de facilitar a integração com serviços externos.

Este artigo está organizado da seguinte forma. Na Seção 2, a Cronos IDE é apresentada, assim como sua arquitetura e implementação. A avaliação da IDE é descrita na Seção 3. A Seção 4 é dedicada aos trabalhos relacionados e, finalmente, a Seção 5 apresenta as conclusões do trabalho.

2. Cronos IDE

2.1. Visão Geral

A Cronos IDE é uma ferramenta para o desenvolvimento rápido de aplicações Java Web. A Cronos permite a criação da aplicação, execução e implantação no ambiente de computação em nuvem sem a necessidade de instalação ou configuração adicional. Além disso, possui suporte para controle de versão, testes unitários, integração contínua e integração com serviços externos providos por terceiros. A Cronos IDE possibilita o desenvolvimento de aplicações para a nuvem baseada no modelo *What You See Is What You Get* (WYSIWYG). Desta forma são disponibilizados elementos visuais para a construção de interfaces gráfica ricas via *Drag and Drop*.

O desenvolvimento no Cronos IDE é baseado na geração de código a partir da modelagem de dados. Com base no modelo de dados fornecido, a IDE define o diagrama de dados e gera um conjunto de classes, que são acessadas por meio de serviços web que seguem o estilo arquitetônico REST (*Representational State Transfer*). Além destes serviços, são disponibilizados mecanismos de autorização, autenticação e uma camada de persistência de dados. A camada web é construída utilizando tecnologias Web tradicionais, tais como HTML, HTML5 e AngularJS.

As principais características da Cronos IDE são: (i) **incentivo à produtividade** – devido ao processo de automatização, parte do código fonte da aplicação é gerado de forma automática (*back-end* e *front-end*) o que tende a aumentar a produtividade da equipe; (ii) **baixa curva de aprendizagem** – devido a Cornos IDE adotar tecnologias padrão no domínio de desenvolvimento para web, a sua curva de aprendizado tende a ser baixa; (iii) **tudo na nuvem** – tanto a IDE quanto a aplicação em desenvolvimento ficam na nuvem, evitando a necessidade de configurações adicionais para os ambientes de desenvolvimento e execução; (iv) **trabalho em equipe** – a Cronos IDE provê suporte ao trabalho colaborativo em projetos de software; (v) **integração com serviços externos** – a Cronos IDE permite a integração de serviços providos por terceiros às aplicações em desenvolvimento.

A Figura 1 apresenta uma visão geral do fluxo de funcionamento da Cronos IDE. Inicialmente, deve-se modelar um diagrama de entidades (*.umlcd) definindo as classes e seus relacionamentos. Em seguida, é gerada o *back-end* da aplicação, representado por um conjunto de classes. Estas classes são denominadas de *entities*. Por sua vez, pode-se gerar o *front-end*, que compreende a parte de interface gráfica com o usuário, e, assim, tem-se a aplicação web completa.

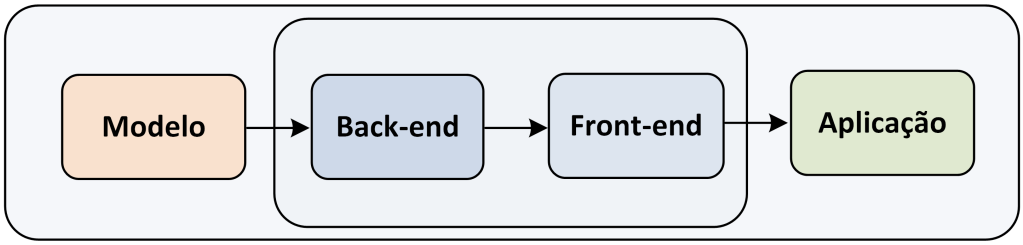


Figura 1. Fluxo de Funcionamento da Cronos IDE.

Back-End – A modularidade auxilia a construção e o gerenciamento de sistemas que envolvem diferentes partes, característica presente em ambientes de computação em

nuvem. A adoção do desenvolvimento baseado em padrões de projeto e serviços web auxilia na modularidade das aplicações desenvolvidas na Cronos IDE. Com as entidades definidas, a aplicação é gerada conforme a estrutura de pacotes a seguir:

- **Pacote de Entidades** – para cada entidade desenhada no diagrama de classes de persistência será gerada uma classe do com a anotação `@Entity`. Essas classes de entidade seguem o formato de POJO (*Plain Old Java Object*);
- **Pacote DAO** (*Data Access Object*) – nesse pacote são geradas as classes de persistência que fazem a manipulação dos dados e a comunicação com o banco de dados. Para cada classe de entidade uma classe DAO, com métodos próprios para a persistência da entidade correspondente, é gerada. A classe `BasicDAO` é gerada com os métodos comuns a todas as classes do tipo DAO. A classe `SessionManager` implementa os métodos de comunicação com o banco de dados;
- **Pacote Business** – nesse pacote são geradas as classes, uma para cada entidade, que implementam as regras de negócios da aplicação;
- **Pacote REST** – nesse pacote são geradas as classes que implementam os `EndPoints` dos serviços REST para as operações CRUD (*Create, Read, Update e Delete*) de cada uma das entidades. Na classe *RESTApplication* é configurado o `EndPoint` de acesso aos serviços REST. Esse `EndPoint` é informado no parâmetro `path` do diálogo de geração da camada de persistência;
- **Pacote Test** – nesse pacote são geradas as classes *stub* para a implementação dos testes unitários da camada de persistência;
- **Pacote Security** – nesse pacote são geradas classes *stub* para lidar com questões de segurança.

Front-End – A partir do diagrama de entidades são geradas as páginas em AngularJS para realizar as operações CRUD para cada uma das entidades. O *front-end* pode ser alterado por meio de um editor *drag and drop*. A Figura 2 apresenta a tela principal da Cronos IDE.

Funcionalidades Adicionais – A Cronos IDE possui um conjunto de ferramentas para auxiliar no desenvolvimento, destacando-se um editor de internacionalização, gerenciador de SQL e gerador de relatórios Ad-hoc.

Além da construção das aplicações, a Cronos IDE também permite construir serviços denominados *gluonsofts*. Os *gluonsofts* são componentes autossuficientes de software, próprios para rodar na nuvem e podem ser acoplados para montar uma aplicação que resolva um problema de negócio complexo ou podem ser utilizados individualmente. Um *gluonsoft* pode ser um serviço que emite nota fiscal eletrônica para uma pequena ou micro empresa, por exemplo.

A partir da Cronos IDE, a implantação da aplicação gerada pode ser realizada diretamente em soluções baseadas na plataforma Cloud Foundry [CF 2016], tais como EMC Pivotal, IBM BlueMix e HP Helios. Além disso, pode-se agendar a publicação contínua a partir de um repositório do código com intervalo de 5 minutos, por exemplo.

2.2. Arquitetura

A Cronos IDE foi desenvolvido com base no *Eclipse Rich Client Platform* (RCP) e no *Eclipse Remote Application Platform* (RAP). A arquitetura da Cronos IDE foi concebida em camadas, conforme apresenta a Figura 3. Cada uma destas camadas é descrita a seguir:

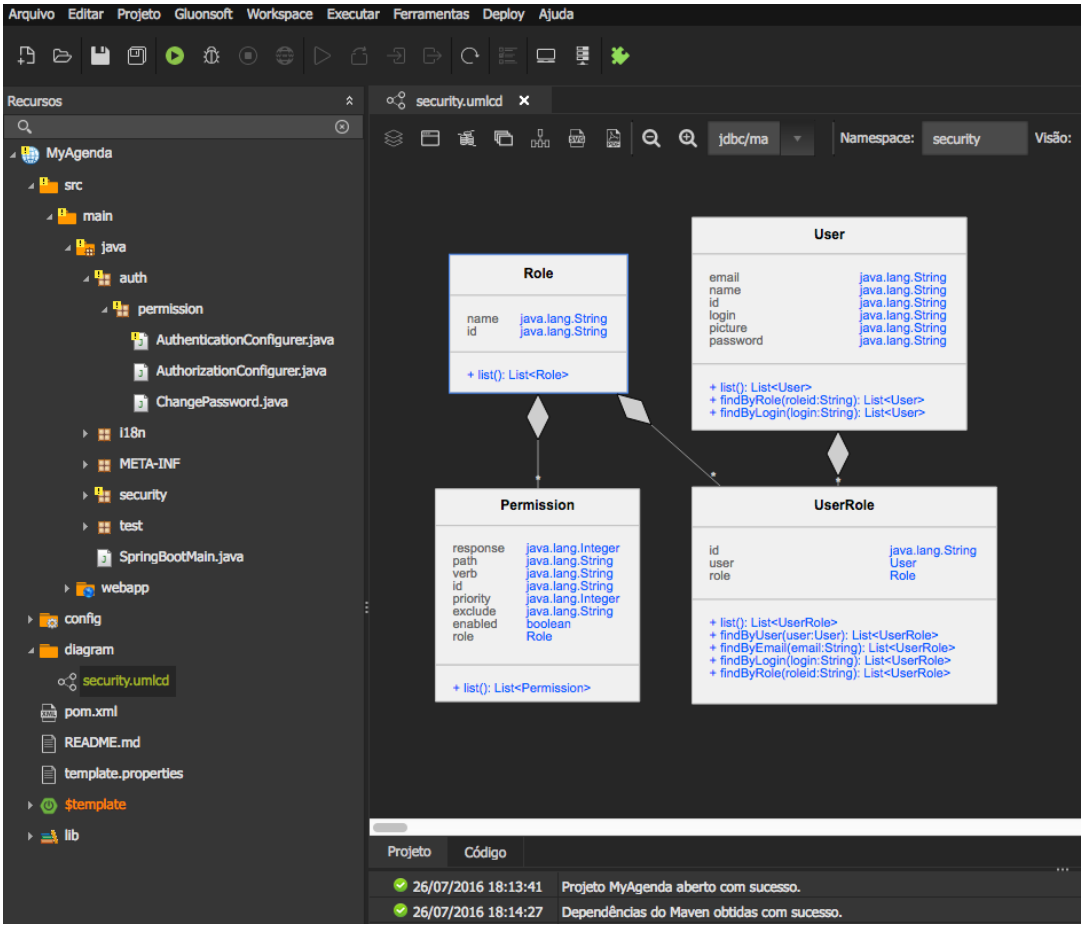


Figura 2. Tela Principal da Cronos IDE.

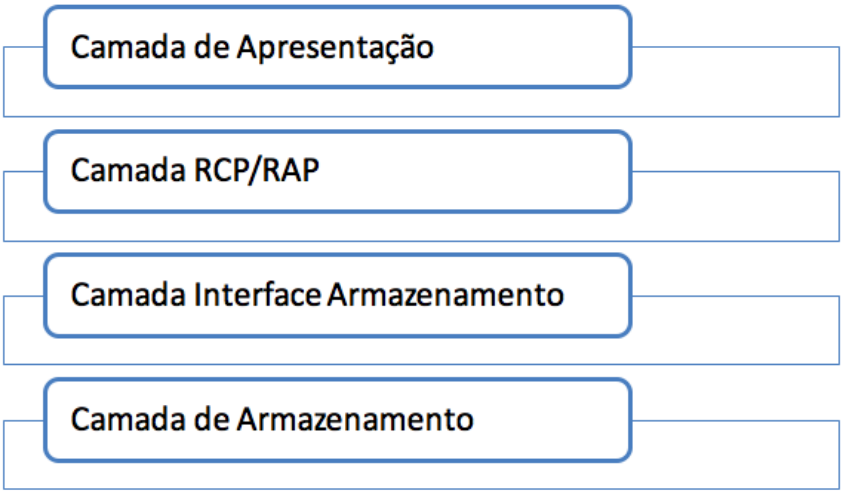


Figura 3. Arquitetura da Cronos IDE.

- **Camada de Apresentação** – essa camada faz a interface com o usuário e é representada pelo RAP Client;
- **Camada RCP/RAP** – essa camada representa a parte servidora da Cornos IDE;

- **Camada de Interface de Armazenamento** – essa camada serve para abstrair as funcionalidade da camada de armazenamento de forma a ser representada como um disco virtual privado. Essa camada engloba o sistema de controle de versão e a área destinada ao *workspace* do usuário;
- **Camada de Armazenamento** – essa camada representa a persistência do *workspace* do desenvolvedor, onde são armazenados os projetos desenvolvidos.

2.3. Implementação

A Cronos IDE foi desenvolvida com Java 8 utilizando o Eclipse RAP/RCP e o Java Development Tools (JDT), ambos do Projeto Eclipse. O Editor Ace foi utilizado como base para a construção da parte de edição. A persistência foi desenvolvida com base na Java Persistence API (JPA) seguindo as implementações EclipseLink e Hibernate conectadas ao SGBD MySQL. Para a construção dos *gluonsofts* foi utilizada a linguagem RESTful API Modeling Language (RAML). A Cronos IDE é executada na infraestrutura Amazon AWS.

3. Estudo de Caso

A avaliação foi realizada por meio de um estudo de caso com um grupo de 10 alunos da disciplina de Desenvolvimento de Aplicações Web ministrada em 2015.2 no Curso de Engenharia da Computação da Universidade Federal do Ceará. Vale ressaltar que apenas alunos que não possuíam experiência prévia com desenvolvimento web participaram da avaliação (i.e., todos os conhecimentos necessários foram adquiridos durante a disciplina, que foi ministrada utilizando Java e tecnologias relacionadas).

O objetivo foi desenvolver uma aplicação de agenda simples, *MyAgenda*. Essa aplicação permite que o usuário faça autenticação e realize operações de cadastro, edição, seleção e remoção de contatos, os quais possuem informações sobre nome, email e telefone. Para comparar o uso da Cronos IDE, o grupo de alunos foi dividido em dois subgrupos com 5 alunos cada: os alunos do subgrupo A, que utilizaram apenas os conhecimentos adquiridos ao longo da disciplina, e os alunos do subgrupo B, que foram designados para utilizar a Cronos IDE. Os alunos do subgrupo B tiveram acesso à documentação da Cronos IDE durante o período de 90 minutos, visto que estes alunos não possuíam conhecimento prévio sobre a IDE.

Para o desenvolvimento da *MyAgenda*, os alunos do subgrupo A gastaram em média 180 minutos. Apesar de ser uma aplicação simples, os alunos precisaram realizar algumas configurações, principalmente da camada de persistência, autenticação e *front-end*. Já os alunos do subgrupo B gastaram em média 30 minutos para desenvolver a aplicação. Isso se deve a ausência da necessidade de configurações, visto que é necessário apenas o navegador web e as funcionalidades da IDE, principalmente a geração do *back-end* e *front-end* a partir da modelagem de dados.

Além disso, o código gerado segue padrões de projeto, facilitando a manutenção e o reuso. Adicionalmente, o editor de *front-end* permite realizar ajustes de forma rápida e simples na interface com o usuário. Além da aplicação *MyAgenda*, os alunos do subgrupo B responderam a um questionário. De acordo com as respostas deste questionário, 80% dos alunos afirmaram que a Cronos IDE tornou o desenvolvimento mais rápido e fácil. Por outro lado, a maioria dos alunos afirmou que a usabilidade da IDE é boa, mas que poderia ser melhorada assim como a documentação atual.

Embora seja uma avaliação inicial, os resultados obtidos são animadores. Por exemplo, o tempo médio de desenvolvimento gasto pelo subgrupo B dá indícios de que a Cronos IDE atende aos requisitos de melhoria da produtividade e baixa curva de aprendizado. Além disso, vale ressaltar que a Cronos IDE já está sendo utilizada para o desenvolvimento dos sistemas da empresa Techne, responsável pelo projeto da IDE.

Como a Cronos IDE é uma aplicação Web, a ferramenta está acessível no domínio `http://ide.cronospaas.com.br`. A documentação está disponível em `http://docs.cronospaas.com` e na seção "Primeiros Passos", pode-se encontrar diversas telas da IDE. Por estar em fase de testes, a IDE está disponível apenas em horário comercial.

4. Trabalhos Relacionados

Uma IDE em nuvem deve prover o máximo de recursos possível, no entanto não pode aprisionar os seus usuários em uma tecnologia proprietária, permitindo a movimentação das aplicações construídas para outros ambientes se necessário. Existem IDE próprias para desenvolvimento de aplicativos na nuvem, que se integram com PaaS (*Platform as a Service*) para hospedagem desses aplicativos, tais como Cloud9 [Cloud9 2016] e Nitrous [Nitrous 2016].

Em [Aho et al. 2011] os autores apresentam a IDE Arvue, uma ferramenta para o desenvolvimento e a publicação de aplicações web. Esta ferramenta apresenta controle de versão e integração com IaaS (*Infrastructure as a Service*) para a hospedagem. Já [Mutiara et al. 2014] apresentam uma pesquisa sobre os requisitos para a construção de uma IDE para nuvem, destacando alguns testes no ambiente de rede local e um protótipo de ferramenta. Contudo, ambas as ferramentas não geram o *back-end* e *front-end* para a aplicação.

As empresas [Mendix 2016] e [OutSystems 2016] apresentam IDEs proprietárias para o desenvolvimento de aplicações para nuvem. A Fundação Eclipse também tem apresentado uma iniciativa nesta direção por meio do projeto Eclipse Che [Eclipse 2016] que tem como objetivo prover uma IDE na nuvem com as funcionalidades semelhantes ao IDE Eclipse padrão.

Um comparativo entre os ambientes para programação em nuvem pode ser encontrado em [Fylaktopoulos 2016]. Segundo os autores, um ambiente de programação moderna deve incluir ferramentas que ajudam a equipe a cumprir todas as fases do ciclo de vida do desenvolvimento de software. Os autores sugerem uma completa mudança de paradigma, talvez seguindo a linha de Engenharia Dirigida por Modelo (do inglês, *Model-Driven Engineering*), pelo fato de aumentar o nível de abstração permitido pelas linguagens de programação de terceira geração, e, também, por permitir ir além do modelo orientado a objetos que, segundo os autores, já alcançou seu ponto de exaustão.

Diferente das abordagens apresentadas e buscando criar um novo paradigma, a Cronos IDE está sendo desenvolvida focada na produtividade obtida por meio da geração das *back-end* e *front-end* a partir de um modelo de dados. Além disso, a Cronos IDE faz uso extensivo de tecnologias *open-source* amplamente utilizadas no mercado, evitando assim o *lock-in*. Adicionalmente, a Cronos IDE permite a criação rápida de *gluonsofts*, acelerando o desenvolvimento e o melhorando o reuso de software.

5. Licença

As funcionalidades da ferramenta aqui descrita poderão ser utilizados por meio de planos gratuitos e/ou pagos.

6. Conclusão

Este trabalho apresentou a Cronos IDE, uma ferramenta web para o desenvolvimento de aplicações Java na nuvem. A Cronos IDE utiliza um editor gráfico para facilitar a construção das aplicações. Avaliou-se a Cronos IDE por meio de um caso de uso. Pela análise dos resultados obtidos, foi possível verificar os ganhos de produtividade no desenvolvimento de software para a nuvem. Como trabalhos futuros pretende-se realizar novos estudos de casos considerando outros cenários para avaliar melhor a Cronos IDE, assim como comparar com outras IDEs Web. Em seguida, pretende-se adicionar nosso novo modelo tarifação [Ribas et al. 2016] à Cronos IDE, permitindo uma cobrança adequada ao uso da ferramenta. Por fim, pretende-se adicionar suporte para a técnica de multi-inquilino e testes de usabilidade e stress, visto que o ambiente em nuvem apresenta muitas variações no desempenho.

Agradecimentos

Esta pesquisa é parcialmente apoiada pela Financiadora de Estudos e Projetos - FINEP (Proc. no. 03-13-0433-00).

Referências

Aho, T., Ashraf, A., Englund, M., Katajamäki, J., Koskinen, J., Lautamäki, J., Nieminen, A., Porres, I., and Turunen, I. (2011). Designing ide as a service. *Communications of Cloud Software*, 1:1–10.

CF (2016). *Cloud Foundry*. `https://www.cloudfoundry.org/`.

Cloud9 (2016). *Cloud9*. `https://c9.io/`.

Eclipse (2016). *Next-Generation Eclipse IDE*. `https://eclipse.org/che/`.

Fylaktopoulos, G. e. a. (2016). An overview of platforms for cloud based development. *SpringerPlus*, 5:1–13.

Mendix (2016). *Mendix*. `https://www.mendix.com`.

Mutiara, A. B., Refianti, R., and Witono, B. A. (2014). Developing a saas-cloud integrated development environment (IDE) for c, c++, and java. *CoRR*, abs/1411.5161.

Nitrous (2016). *Nitrous*. `https://www.nitrous.io/`.

OutSystems (2016). *OutSystems*. `https://www.outsystems.com`.

Ribas, M., Lima, A. S., de Souza, J. N., Sousa, F. R. C., and Moreira, L. O. (2016). Um modelo para gerenciamento de bilhetagem paas em ambientes de computação em nuvem. *Revista IEEE América Latina*, 14.

Sousa, F. R. C. and Machado, J. C. (2014). Gerenciamento de dados em nuvem. In *XXXIII Jornadas de Atualização em Informática (JAI 2014)*, pages 1–40.

GuideAutomator: Automated User Manual Generation with Markdown

Allan dos Santos Oliveira[1], Rodrigo Souza[1]

[1] Department of Computer Science – Federal University of Bahia (UFBA) – Salvador – BA – Brazil

allanoliver@dcc.ufba.br, rodrigo@dcc.ufba.br

Abstract. *User manual, also known as user guide, is a technical document for communication designed to assist end users of a product. It aims at filling the gap between what is easily deductible and what is not. A complicating factor for those who write these documents is to keep them up-to-date with changes on the application over time, like addition/removal of features or just changes on user interfaces (documented as screenshots). This work aims at assisting writers of such documents, for web applications only, providing automated screen capture, reducing maintenance cost and user manual inconsistency. We provide GuideAutomator, which enables this automatization through writing of documents under Markdown syntax and encapsulation of Selenium Web Driver.*

https://youtu.be/zXZyNgJOgdY

1. Introduction

User experience has been a rising concern in the software industry, due to its decisive role on user engagement and consequently on software's success. A good user experience design enhances user satisfaction by effectively addressing the needs and circumstances of its users, which is done mainly by providing carefully considered user interfaces.

Even though software user interfaces are getting easier to use, applications still require user manuals for several reasons. Some notorious reasons are: unexperienced users, first time users may face difficulties using the application; domains policy, as some domains formally require user manuals to support its users; support critical decisions, some critical actions like delete resources may rise doubts; reveal full potential of an application, some functionalities might not be explicitly visible nor easily deductible.

Therefore, the user manual is an important part of software documentation. However, development teams often ignore it, primarily because of painful standard tools, such as popular text editors like Microsoft Word[1] and LibreOffice Writer[2]. Those are mentioned as painful because they slow down development speed, make versioning challenging to manage due to binary files, and make it harder to keep software documentation synced with the software's growth, especially as these documents contain many screen capture images.

[1] Microsoft Word. Retrieved July 30 from https://products.office.com/en/word
[2] LibreOffice Writer. Retrieved July 30 from https://www.libreoffice.org/discover/writer/

To address these issues, this work assists designers of user manuals for web applications by providing GuideAutomator, a user manual generation tool with automated screen capture implemented upon Selenium Web Driver[3] (a browser automation framework), to be used with Markdown[4] (a lightweight markup language), which can be easily versioned to improve documentation maintainability.

All user manuals aim at providing clear information to their users about how to use an application. In order to accomplish that, there are some good practices for designing such documents (Hodgson, 2007). These practices may include use of summary, instructions on how to use main functionalities, troubleshooting section, glossary, etc. The most important parts, which are the instructions, uses several techniques to facilitate user assistance, like providing a systematic sequence to accomplish a certain task. To support that an effective approach is to make use of screen capture.

Screen captures or screenshots, under this context, are images taken from an application to reproduce either full or part of a user interface in a given application state. It reduces reasoning time for users to find or execute a function and this is the desired scenario to all stakeholders. GuideAutomator comes in to bring automation to this process of screen capture, alongside with Markdown writing as described in the next section.

The remainder of this paper is divided into 5 sections. In section 2 we go through GuideAutomator's description and operation. Section 3 presents the related work. Section 4 draws the conclusion and future work. Finally, Section 5 lists the references for this work.

2. An automation tool

GuideAutomator is a command line Node.js application, released under MIT license, which takes as input a mix of a widely used lightweight markup language, Markdown, with GuideAutomator's API for performing actions on web browsers. Our API is a wrapper for a browser automation tool, Selenium Web Driver. Figure 1 presents the application workflow. Essentially, a user manual writer composes the document with Markdown's rich text as in any common Markdown document. However, to reproduce user steps and capture the outcome stages of these steps, the writer must also include our API commands into the Markdown text. Our application processes the API commands and the user manual can be then outputted to PDF and HTML formats.

GuideAutomator runs on Windows, Linux and Mac OS platforms, as long as it has Node.js installed. Currently, it only supports Chrome Browser with its respective web driver, ChromeDriver, as discussed on the Selenium Web Driver subsection. The application is available at NPM[5], the default package manager for Node.js, under the

[3] Selenium WebDriver. Retrieved April 17, 2016 from http://www.seleniumhq.org/projects/webdriver

[4] Gruber, John. Markdown Syntax Documentation. Retrieved April 17, 2016 from https://daringfireball.net/projects/markdown/syntax

[5] npm, Inc. (2016). guide-automator. Retrieved May 22 from https://www.npmjs.com/package/guide-automator

name of "guide-automator" and runs as the example shown on Figure 2. The application takes two parameters: input file path (Markdown mixed with GuideAutomator's API) and output folder path (for placing generated documents).

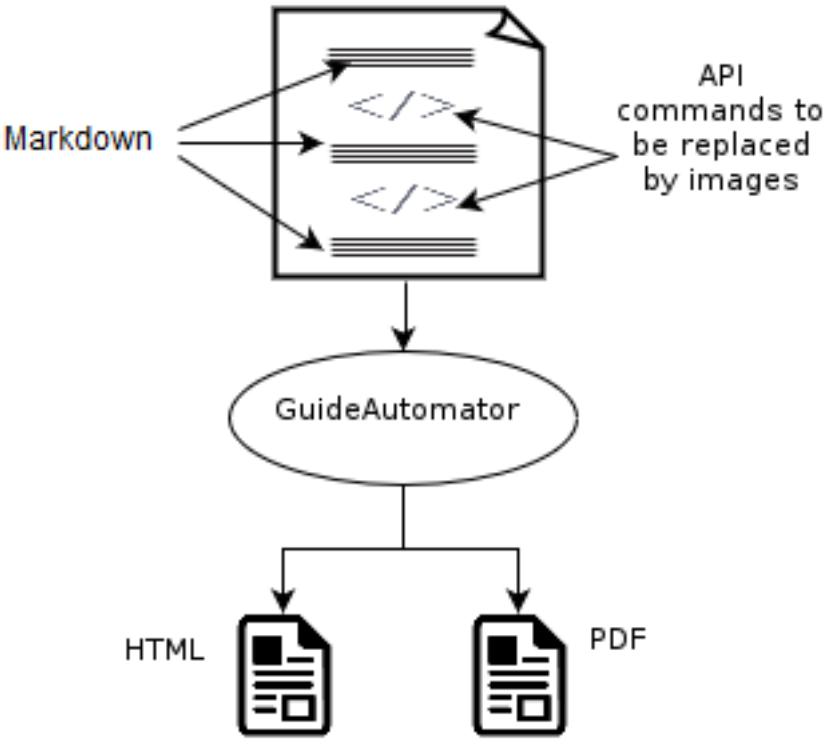


Figure 1. GuideAutomator's workflow.

```
guide-automator docs\input.md docs
```

Figure 2. Example for running GuideAutomator.

2.1. **Markdown**

Markdown is a plain text formatting syntax created in 2004 by John Gruber, originally designed to be converted to valid XHTML or HTML. Over the last decade, many extensions were created making it possible to convert Markdown's language to many other formats, like PDF, LaTeX, RTF, etc. Alongside with all these possibilities, its ease of use has made Markdown very popular in the software community. Some popular applications that make use of Markdown are GitHub[6] and StackOverflow[7]. These applications and many others use either standard Markdown syntax or a custom extension of the original markup language.

[6] GitHub Inc. About writing and formatting on GitHub. Retrieved April 17, 2016 from https://help.github.com/articles/about-writing-and-formatting-on-github/

[7] Stack Overflow Inc. How do I format my posts using Markdown or HTML? Retrieved April 17, 2016 from http://stackoverflow.com/help/formatting

Markdown was intended to be simple, easy-to-read and easy-to-write. Its syntax is composed by regular text with a few non-alphabetic characters, which were chosen to look like what they mean. Some of the non-alphabetic characters are "#" (hash) and "*" (asterisk), which are used to style headers and to define unordered lists, respectively. A complete syntax definition is available at the official Markdown page by John Gruber.

2.2. **Selenium Web Driver**

To make automation possible, GuideAutomator encapsulates Selenium Web Driver, a powerful tool for browser automation, which makes it possible to drive a browser natively as a user would. It supports of some of the largest browser vendors, such as Firefox, Chrome, Safari and Internet Explorer. However, due to a few unsupported operations (take screenshot of viewport only and take screenshot of an element) for some webdrivers, GuideAutomator v1.0 guarantees support only to ChromeDriver 2.21[8]. As webdrivers expand their support to required operations, GuideAutomator will expand support to more web browsers.

Selenium Web Driver provides a rich API for manipulating a web browser. Major commands and operations are: fetching pages, locating UI (User Interface) elements, filling in forms and taking screenshots. GuideAutomator's API incorporates these and many other commands as described in the following subsection.

2.3. **API**

GuideAutomator provides an API for performing common behavior of web applications users, like clicking elements, filling in inputs, etc. Alongside these actions, GuideAutomator's API provides a command for taking screenshots, so these images can automatically be placed on the user manual, regardless of the output format.

For calling API's commands, the user manual writer must include on their Markdown file blocks of GuideAutomator's code, bounded by specific delimiters, defined as `<automator>` for block code beginning and `</automator>` for block code end, see example on Figure 3. Commands must be separated by special character ";" (semicolon), as shown in Figure 2. GuideAutomator can then compile those commands, and, in case of any failure, it outputs the incorrect token. Otherwise, the application proceeds with processing those commands. Table 1 shows the current list of available API commands for GuideAutomator v1.0.

Once all commands are executed, GuideAutomator gathers all generated images and places them on their appropriate location in the outcome document. Appropriate image locations on the outcome document are defined based on the location of their respective `<automator>` code block, i.e., the code block that encloses their respective `takeScreenshot` or `takeScreenshotOf` operations. This process can be repeated as much as needed to keep a user manual up-to-date with its respective application, as every time GuideAutomator is executed it generates all screen captures all

[8] Google Inc. ChromeDriver - WebDriver for Chrome. Retrieved April 17, 2016 from https://sites.google.com/a/chromium.org/chromedriver/

over again. Figure 3 and 4 illustrate, respectively, the input file for logging into Yii Blog Demo[9] and the output HMTL page containing generated screenshots. Note that captured images are placed where their respective block commands were.

Command	Description
get(url)	Navigates to the page with the specified *url*.
takeScreenshot	Takes screenshot of the browser's viewport.
takeScreenshotOf(selector)	Takes screenshot of the browser's viewport after the element identified by the css *selector* has been scrolled into view.
fillIn(selector,content)	Types *content* on the element identified by the css *selector*.
submit(selector)	Submits the form containing the element identified by the css *selector*, if there is one.
click(selector)	Performs a click on the element identified by the css *selector*.

Table 1. GuideAutomator's API commands, version 1.0.0.

input.md

```
# How to log into Yii Blog Demo
## First, access http://www.yiiframework.com/demos/blog/
<automator>
get('http://www.yiiframework.com/demos/blog/');
takeScreenshot;
</automator>

## Click on the Login tab
## Fill in the login form with valid username and password
## Submit the login form
<automator>
click('#mainmenu li:last-child > a');
fillIn('#LoginForm_username','demo');
fillIn('#LoginForm_password','demo');
takeScreenshotOf('#login-form');
</automator>

# Now, feel free to use the dashboard
<automator>
submit('#login-form');
takeScreenshot;
</automator>
```

Figure 3. Simple input example for logging into Yii Blog Demo.

[9] Yii Software LLC. Yii Blog Demo. Retrieved May 20, 2016 from http://www.yiiframework.com/demos/blog/index.php/post/index

How to log into Yii Blog Demo

First, access http://www.yiiframework.com/demos/blog/

Click on the Login tab

Fill in the login form with valid username and password

Submit the login form

Now, feel free to use the dashboard

Figure 4. Generated web page from example in Figure 3.

3. Related Work

GuideAutomator has its roots in approaches such as literate programming, automatic documentation generators, standardized markup languages, continuous integration, and functional testing. We comment related work on these topics below.

Waits and Yankel (2014) claim that standard documentation tools and processes, based on the collaborative writing of documents using file formats that are binary and proprietary, such as Microsoft Word documents, are not well integrated into software development tools and processes. The main problems of the current approach are the difficulty of merging contributions of different authors, due to limitations of version control systems, and the inconsistency of document presentation across operating systems. They propose writing documentation in plain text files, using a standardized markup language such as Markdown, and then convert it into formats such as PDF and HTML using a tool such as Pandoc[10]. GuideAutomator follows this approach and further automates the process by generating images from user interface scripts interleaved with text in the documentation.

Literate programming is an approach to programming in which the source code is interleaved with text explaining the internals of a program (Knuth, 1984). The focus is on the documentation, and the source code is presented in parts following the structure of the documentation. This document can be transformed either into pure source code, which can be compiled and executed, or into documentation. The approach is currently being used in reproducible research (Madeyski, 2015). Our tool is also based on documentation interleaved with source code. In our case, however, the text is not intended to explain the source code; instead, it explains the system to end users, and the source code is used to generate images that help explain the text.

API documentation generators are tools that extract structured comments in the source code of a program and generate the documentation of its application programming interface (API), aimed at programmers. API documentation generators include Doxygen[11] and Javadoc[12]. GuideAutomator is also an automated documentation generator; however, the documentation is geared towards end users, not programmers.

Continuous integration is the practice of checking in developers' source code changes to a shared repository frequently (Fowler, 2006). Following a check-in, an automated build process compiles the source code and possibly perform other automated tasks, such as running unit tests, performing static analysis, and even deploying the application to servers (in this case, the approach is known as continuous deployment). Because GuideAutomator automates the task of generating user guides containing screen captures, it can be included in the continuous integration cycle and enable the creation and publication of documentation that is in sync with the latest source code change.

---

[10] John MacFarlane. Pandoc: a universal document converter. Retrieved May 16, 2016 from http://pandoc.org

[11] Dimitri van Heesch. Doxygen. Retrieved May 16, 2016 from http://www.doxygen.org

[12] Oracle Corporation. Javadoc. Retrieved May 16, 2016 from http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

Functional testing is the process checking whether the behavior of a software application conforms to its specification, from the end-user point of view, by executing test cases derived from its specification (Myers et al., 2011). For web applications, functional testing can be automated using the framework Selenium (Holmes and Kellogg, 2006). GuideAutomator uses Selenium for a different purpose; instead of comparing the application's output with a specification, it captures the output so it can be displayed in a user manual.

4. Conclusion

Development teams often ignore user documentation as a direct result of the pain standard documentation tools and processes cause (Waits, 2014). To overcome this issue, GuideAutomator comes in as an easy to write and maintain automation tool for generating user manuals. Its ease of use and maintenance is due to a high level API for driving a web browser and taking screenshots, and to its plain text syntax that can be even versioned alongside the main project source code.

Screen capture automation dramatically reduces writers' effort. When the application changes, one will not need to update every single image in the user manual as those images will be captured once GuideAutomator is executed to represent the current application state.

As future work, we intend to expand support for more web browsers and create more API commands for browser manipulation. In addition, to provide a better experience to customers, it is our intention to expand options for taking screenshot to allow, for instance, image customization like highlights, borders, resizing, etc. Because we are on the early stages of development, there have been no evaluation of GuideAutomator yet. Therefore, evaluation on real projects is also on the roadmap, so we will be able to validate GuideAutomator's benefits in practical usage.

5. References

Hodgson, P. (2007). Tips for writing user manuals. Retrieved April 17, 2016 from http://www.userfocus.co.uk/articles/usermanuals.html

Waits, T. and Yankel, J. (2014). Continuos System and User Documentation Integration.

Knuth, D. (1984). Literate Programming. In *The Computer Journal*, Vol. 27, No. 2, 97-111.

Madeyski, L. and Kitchenham, B.A. (2015). Reproducible Research–What, Why and How. *Wroclaw University of Technology, PRE W, 8*.

Fowler, M. (2006). Continuous Integration. Retrieved May 16, 2016 from http://martinfowler.com/articles/continuousIntegration.html

Myers, G., Sandler, C. and Badgett, T. (2011). The Art of Software Testing, 3rd edition. Wiley, New York, NY.

DawnCC : a Source-to-Source Automatic Parallelizer of C and C++ Programs

Breno Campos Ferreira Guimarães, Gleison Souza Diniz Mendonça, Fernando Magno Quintão Pereira

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{brenosfg,gleison.mendonca,fernando}@dcc.ufmg.br

Abstract. *Dedicated graphics processing chips have become a standard component in most modern systems, making their powerful parallel computing capabilities more accessible to developers. Amongst the tools created to aid programmers in the task of parallelizing applications, directive-based standards are some of the most widely used. These standards, such as OpenACC and OpenMP, facilitate the conversion of sequential programs into parallel ones with minimum human intervention. However, inserting pragmas into production code is a difficult and error-prone task, often requiring familiarity with the target program. This difficulty restricts the ability of developers to annotate code that they have not written themselves. This paper describes DawnCC, a tool that solves this problem. DawnCC is a source-to-source compiler module that automatically annotates sequential C code with OpenACC or OpenMP directives; thus, effectively producing parallel programs out of sequential semantics. DawnCC is equipped with a number of static program analyses that: (i) infer bounds of memory regions referenced in source code to copy them between host and device; and (ii) discover parallel loops in sequential code. To validate its effectiveness, we have used DawnCC to automatically annotate the benchmarks in the Polybench/GPU suite with proper OpenACC directives. These annotations let us parallelize these benchmarks, leading to speedups of up to 78x.*

Link to Video: https://youtu.be/e7mmpP3x10E

1. Introduction

The growing popularity of heterogeneous architectures containing both CPUs and GPUs has generated an increasing interest in general-purpose computing on graphics processing units (GPGPU) [**?**]. This practice consists of developing general purpose programs, *i.e.* not necessarily related to graphics processing, to run on hardware that is specialized for graphics computing. Executing programs on such chips can be advantageous due to the parallel nature of their architecture: while a typical CPU is composed of a small number of cores capable of a wide variety of computations, GPUs usually contain hundreds of simpler processors, which perform computations in separate chunks of memory concurrently [**?**]. Thus, a graphics chip can run programs that are sufficiently parallel much faster than a CPU [**?**]. In some cases, this speedup can reach several orders of magnitude. GPUs can also be not only faster, but also more energy-efficient when running memory-parallel tasks.

This model, however, has its shortcomings. Historically, parallel programming has been a difficult paradigm to adopt, sometimes requiring that developers be familiarized with particular instruction sets of different graphics chips. Recently, a few standards such as OpenCL and CUDA have been designed to provide some level of abstraction to these platforms, which in turn has led to the development of compiler directive-based programming models, *e.g.* OpenACC [**?**] and OpenMP [**?**]. While these have somewhat bridged the gap between programmers and parallel programming interfaces, they still rely on manual insertion of compiler directives, an error-prone process that also commands in-depth knowledge of the target program.

Amongst the hurdles involved in annotating code, two tasks are particularly challenging: identifying parallel loops and estimating memory bounds [**?**]. Regarding the former, opportunities for parallelism are usually buried under complex syntax. As to the latter, languages such as C and C++ do not provide any information on the size of memory being accessed during the execution of the program. However, when offloading code for parallel execution, it is necessary to inform which chunks of memory must be copied to other devices. Therefore, the onus of keeping track of these memory bounds falls on the programmer.

This paper describes DawnCC , a tool that we have designed, implemented and tested to shield developers from the complexities of parallel programming. Through the implementation of a static analysis that derives memory access bounds from source code, it infers the size of memory regions in C programs. With these bounds, our tool is capable of inserting data copy directives in the original source code. These directives provide a compatible compiler with information on which data must be moved between devices. It is also capable of identifying loops that do not contain memory dependences, and therefore can be run in parallel, and marking them as such with the proper pragmas. To increase the amount of potentially parallel loops detectable, it performs pointer disambiguation. That is, it determines conditions that guarantee the absence of aliasing, and indicates that a loop may be executed in parallel when said conditions are met.

We have developed DawnCC as a collection of compiler modules, or passes, for the LLVM compiler infrastructure [**?**]. We have made DawnCC available for public use through a webpage that functions as a front-end[1]. Users can submit their own C source code through this page. The code is then compiled to LLVM bytecode and run through our passes, which analyze it and reconstruct the original C source, inserting the appropriate parallel standard directives. Thus, the user receives as output a modified version of their code in plain text, which corresponds to an effectively parallel version of their submitted program. To demonstrate the effectiveness of DawnCC , in this paper we show how to apply it onto the source code of the benchmarks available in the Polybench/GPU suite[2]. We use DawnCC to annotate the programs in Polybench with OpenACC directives. The modified benchmarks present speedups of up to 78x in execution time, and their results are verified for correctness by comparison with sequential execution.

[1]Our tool is currently available at `http://cuda.dcc.ufmg.br/dawn/`

[2]Polybench's source code is available in several different websites, such as `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`

2. Related Work

We could only design and implement DawnCC because of the emergence of annotation systems for data-parallel systems. These standards aim to simplify the creation of parallel programs by providing an interface for developers to annotate specific regions in source code, indicating that they should run in parallel. Parallel execution can be performed in a variety of ways, such as spread between multiple cores in a single CPU, or through offloading computation to a separate device in heterogeneous architectures. Compilers that support these standards can check for the presence of directives in source code, and generate parallel code for the specific regions annotated, which can in turn be allocated to run on target devices. DawnCC currently supports two standards, OpenACC and OpenMP , but it can easily be extended to support others. We have chosen these standards due to their effectiveness and widespread use in modern parallel programming.

To the best of our knowledge, DawnCC is the only source-to-source compiler that inserts OpenACC or OpenMP annotations in programs automatically. However, there are tools with the same end-goal: to compile C into CUDA without much intervention from developers. A number of optimization frameworks based on the polyhedral model have been used for automatic generation of GPU code [**?**, **?**]. These tools generate GPU code directly, without using annotations. In practice, the symbolic limits generated by such frameworks and the ones generated by the analysis chosen for this work present similar results [**?**], each having specific advantages. For instance, the method applied in this paper can handle non-affine regions of code, while the analyses implemented in polyhedral-based tools usually generate simpler interval expressions, by performing static simplification, which comes at the cost of a higher compilation time.

3. Working Example

Figure 1 shows an example program that can be provided as input in DawnCC 's webpage. The C code shown contains a few loops that exemplify some of the key analyses DawnCC is capable of performing, and exposes some of the main functionalities it provides. The following sections explain these in greater depth. For the output examples in the figures that follow, we have used DawnCC to annotate the source code with OpenACC directives, and configured it to only annotate loops it deems as parallelizable. Note that the original code remains unchanged, having been reconstructed from the compiler's intermediate representation.

3.1. Memory Bound Accuracy

Balancing the overhead of offloading computation to a separate device and the gain in performance from parallel execution is a cornerstone of efficient GPGPU programming. In many cases, the effort spent in performing tasks not related to effective computation, such as copying data or synchronizing execution, counterweighs the advantages of abusing parallelism. This can cause the parallel version of the program to show no significant gain in performance, or even a slowdown, even when the algorithm involved presents ripe opportunities for concurrent execution. Therefore, handling the amount of overhead associated with parallelizing code is vital.

When it comes to measuring memory bounds to perform data copying, a naive analysis could simply measure the total size of memory blocks referenced inside the

```
void example (int *a, int *b, int c) {
    int n[5000];
    int i, j, k;

    /*loop 1*/
    for (i = 0; i < 1000; i++) {
      /*loop 2*/
        for (j = 0; j < 1000; j++) {
            n[i+j] = i*j;
        }
    }

    /*loop 3*/
    for (k = 0; k < 5000; k++) {
        a[k] = b[k]+(k*k);
    }

    /*loop 4*/
    for (k = 0; k < c; k++) {
        a[k] = k*(k+1);
    }
}
```

Figure 1. Example input C code.

loops, and use the entire size as the upper bound in a data copy directive. While correct, such an approach could potentially generate redundant copy instructions, since it is possible for chunks of data which are not used within the loop to be copied back and forth between devices. Our analysis instead calculates the bounds of the memory region that is effectively accessed inside the loop, thus minimizing the amount of potentially redundant computation. Figure 2 (a) highlights the pragmas inserted in the code for the first two loops in the original example. Since the upper limit for the array subscript is at most the sum of the values reachable by both induction variables, it is not necessary to copy the entire array. As a result, the data directive generated contains a more precise value that better reflects the effective memory access limits.

3.2. Treating Pointer Aliasing

There are many reasons that might prevent a given piece of program from being parallelizable. Most of these include memory dependences of some form. In C and C++ code involving dynamically allocated memory regions, one of the main culprits behind such dependences is the possibility of pointer aliasing. That is to say, when a set of instructions accesses memory referenced by multiple pointer values, there is no guarantee that the regions pointed to do not overlap. In such cases, an implicit memory dependence exists, and the possibility of parallelization is typically discarded. Usually, treating these cases statically involves the employment of complex and costly interprocedural pointer analyses.

However, as a by-product of our alias analysis, our tool is capable of performing pointer disambiguation. This means it can infer conditions that ensure the absence of aliasing, in which case the memory dependences do not exist, and parallel execution might be possible. By combining this with conditional compilation directives, we can solve the problem in an elegant and concise way. The conditional directives instruct a compiler to create two different versions of a loop, whose execution is controlled by an aliasing check. Then, during execution, the conditional is evaluated. If the absence of aliasing is confirmed, the loop is executed in parallel. Otherwise, a sequential version is executed instead. Figure 2 (b) shows the pragmas inserted for the code that corresponds to the third loop in the original example. The alias check can be observed immediately before the

pragma directives, and the conditional execution pragmas can be seen associated with the data copy and kernels directives.

3.3. Symbolic Inference

Figure 2 (c) shows the code that corresponds to the fourth loop in the original example. In this case, the scalar value of the variables used as subscripts in the memory accesses in the loop are not predictable in the function's scope. In this case, DawnCC is capable of inferring the proper limits by inserting a series of value checks to determine which value effectively defines the upper bound for the memory accesses performed. It then inserts the proper value in the copy directive. Note that in this case the memory bounds may vary during execution, yet the limits defined remain correct for every execution context. The value checks can be seen immediately above the pragmas. The first pragma inserted is the data directive, with the proper upper bound as its parameter.

```
/*loop 1*/
#pragma acc data pcopy(n[0:1999])
#pragma acc kernels
#pragma acc loop independent
for (i = 0; i < 1000; i++) {
        /*loop 2*/
        #pragma acc loop independent
        for (j = 0; j < 1000; j++) {
                n[i+j] = i*j;
}
```
(a);

```
/*loop 3*/
char RST_AI2 = 0;
RST_AI2 |= !((A + 0 > b + 5000)
|| (b + 0 > a + 5000));
#pragma acc data pcopy(a[0:5000],b[0:
5000]) if(!RST_AI2)
#pragma acc kernels if(!RST_AI2)
#pragma acc loop independent
for (k = 0; k < 5000; k++) {
   a[k] = b[k]+(k*k);
}
```
(b);

```
   /*loop 4*/
   long long int AI3[6];
   AI3[0] = c + -1;
   AI3[1] = 4 * AI3[0];
   AI3[2] = AI3[1] + 4;
   AI3[3] = AI3[2] / 4;
   AI3[4] = (AI[3] > 0);
   AI3[5] = (AI3[4] ? AI3[3] : 0);
   #pragma acc data pcopy(a[0:AI3
[5]])
   #pragma acc kernels
   #pragma acc loop independent
   for (k = 0; k < c; k++) {
       a[k] = k*(k+1);
   }
```
(c);

Figure 2. (a) Pragmas inserted in the first and second loops; (b) Pragmas and alias checks for third loop; (c) Pragmas and value checks for fourth loop.

4. Web Interface

DawnCC is available at `http://cuda.dcc.ufmg.br/dawn/`. This webpage is open to the general public, and it receives, as input, a plain C program. Figure 3 shows the main screen of our webpage. Whoever uses the DawnCC webpage can choose between annotating programs with either OpenACC or OpenMP directives. Users have also the option to display compilation statistics about the annotation process. These statistics include facts such as the number of memory accesses that DawnCC has been able to bound, the number of loops inferred to be parallel, the number of total annotations inserted, etc. When dealing with large programs, users have the option to load them instead of pasting their text into the input window.

Figure 4 shows the output produced by our tool. The annotated program is made available to the user at the window in the lower part of the web interface. The user can also download a complete version of the program, via a link next to the program's text. Whenever users select to display compilation statistics, these numbers are displayed right above the output window. The webpage contains a tutorial about how to use DawnCC , which provides more information to the interested reader.

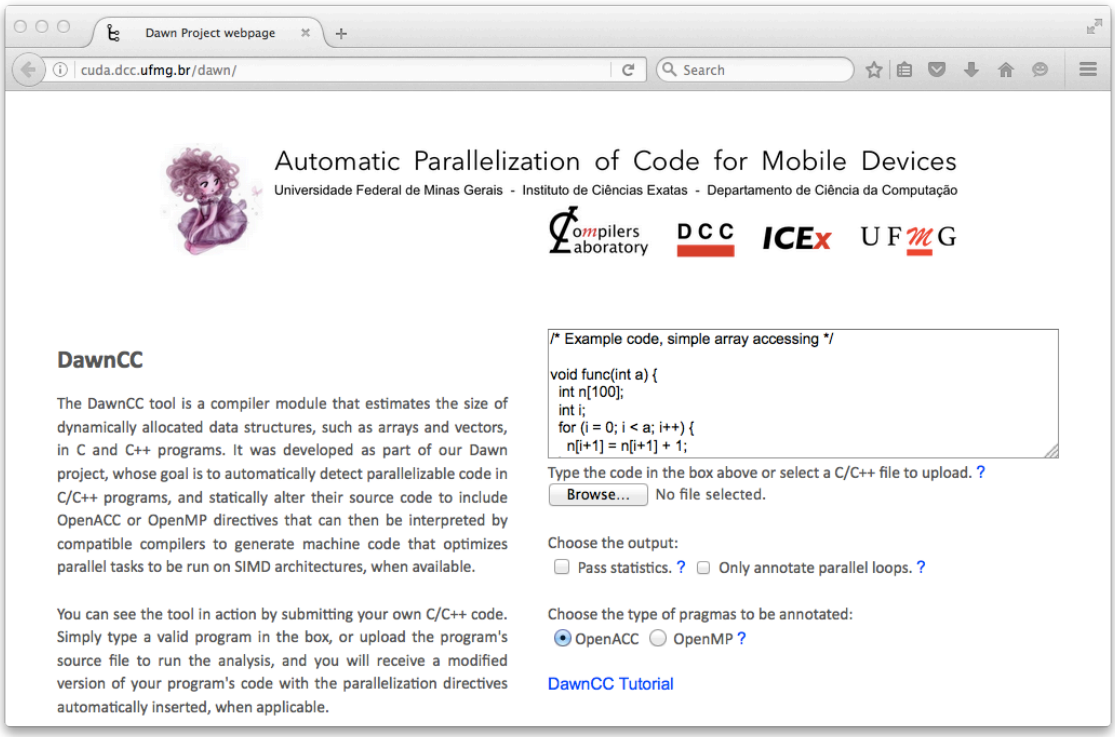


Figure 3. Screenshot of the web interface of DawnCC .

5. Experiments

We performed a small set of tests to validate the effectiveness of DawnCC . We used the programs in the Polybench/GPU suite of benchmarks, contained in the UniBench compilation of suites, as our testing codebase. We used DawnCC to annotate all the programs in the suite with data copy directives and kernels directives. It is important to note that for this specific set of tests the annotation of parallel loops was done manually, as the main focus of this work is the analysis for inferring memory bounds and performing pointer disambiguation. Figure 5 displays a speedup chart that measures the execution time ratio between GPU and CPU execution time. A positive value means a speedup of the given amount was observed, while a negative value corresponds to a slowdown of the same proportion.

The experiments were performed in a server with an Intel Xeon E5-2620 CPU, with 6 cores at 2.00GHz frequency each, and 16 GB of DDR2 RAM. The GPU used for parallel execution was an NVidia GTX 670 with 2GB RAM (CUDA Compute Capability 3.0). All the tests were performed in a Linux Ubuntu 12.04 environment. The compiler used to generate binaries for both baseline and OpenACC-accelerated versions of the benchmarks was Portland Group's PGCC, version 16.1.

Some very significant speedups can be observed in benchmarks that have considerable running time on the CPU, such as Covariance and FDTD-2D benchmarks. In these cases, the CPU took anywhere from 15 to 80 seconds to finish execution, whereas the GPU usually takes under a second. Less significant speedups can be observed in other benchmarks that take a moderate amount of time to execute in the CPU, such as 2MM and SYR2K, which take 5 to 15 seconds to execute. In most benchmarks where

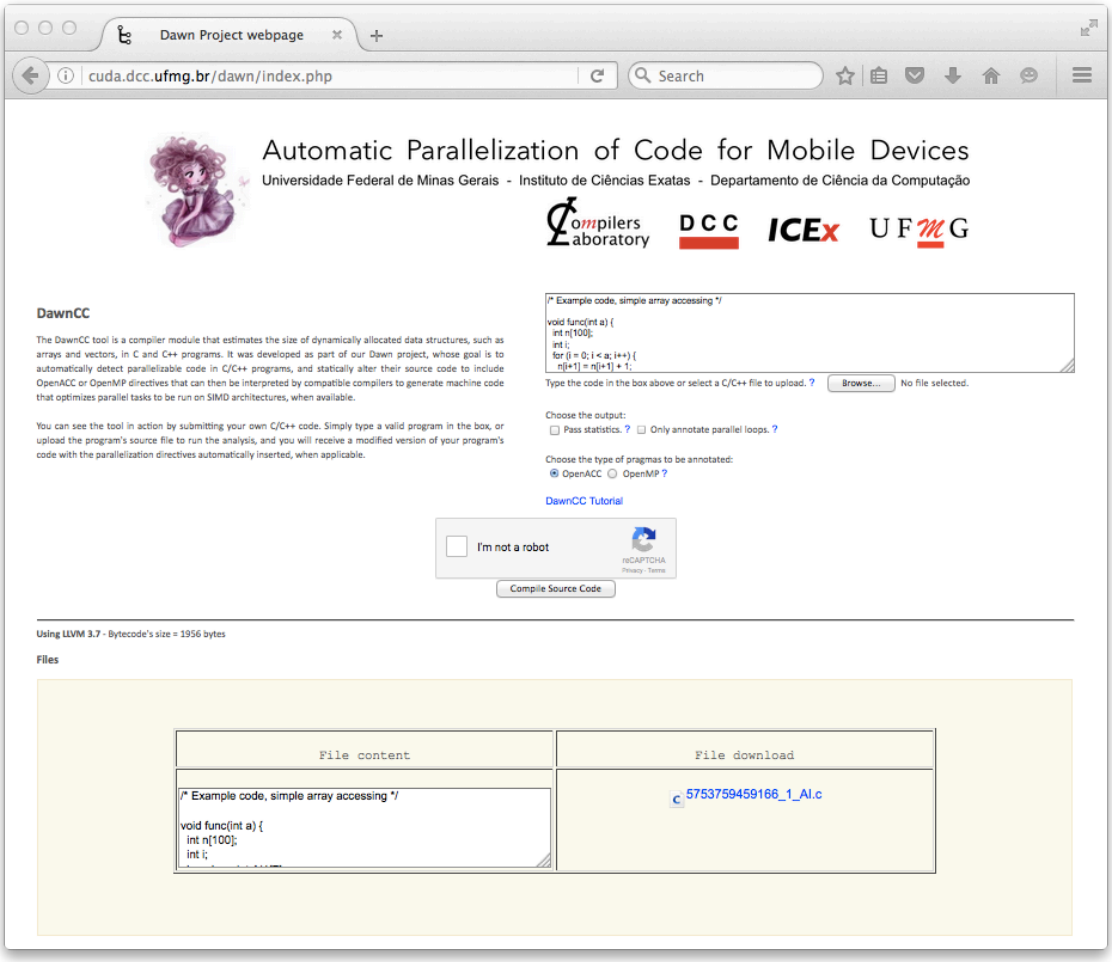


Figure 4. Output window of DawnCC .

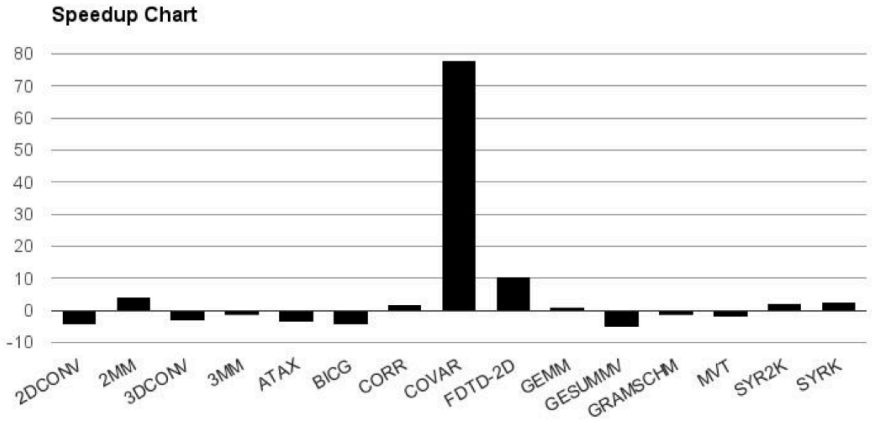


Figure 5. Speedup chart for experiments on Polybench/GPU benchmarks.

slowdowns occurred, the CPU execution time was under 1 second. This indicates that for these cases, the overhead involved in moving memory between devices and offloading execution was more significant than any benefits that parallel execution might have

shown. This could possibly be in part due to the problem sizes used in Polybench/GPU benchmarks. We planned on testing these conjectures empirically by comparing results from different compilers, but OpenACC-compliant compilers are more often than not propietary and expensive, which makes further testing challenging.

6. Conclusion

This paper has described DawnCC , a tool that is currently available at `http://cuda.dcc.ufmg.br/dawn/`. The goal of DawnCC is to facilitate the development of parallel programs that run on Graphics Processing Units (GPUs). Developers can feed this tool with plain C code, and it transforms it into parallel code by annotating loops with either OpenACC or OpenMP 4.0 directives. The key benefit of using DawnCC is performance: annotated code can be as much as 70x faster than their original counterparts. DawnCC still offers room for improvements. In particular, sometimes we perceive slowdowns in a few programs that we annotate using this tool. We are working to remove these slowdowns.

Acknowledgement This work has been sponsored by LG Electronics do Brasil through the project *Automatic Parallelization of Code for Mobile Devices*.

Apoio Computacional para Especificação de Requisitos com Reúso de Padrões de Requisitos

Leonardo Barcelos[1 2], Rosângela Penteado[1]

[1] Departamento de Computação
Universidade Federal de São Carlos (UFSCar) – São Carlos, SP – Brazil

[2] Departamento de Ciências Exatas e da Terra
Universidade do Estado de Minas Gerais (UEMG) – Frutal – MG – Brazil

{leonardo.barcelos, rosangela}@dc.ufscar.br

Abstract. *Studies show that problems associated with the requirements engineering are widely recognized by affect software quality and impact the effectiveness in your development process. The knowledge reuse obtained from previous projects can facilitate the identification and writing of requirements in the elaboration of a complete and consistent requirements document. Software pattern is a solution to capture and reuse knowledge from different contexts at the software development. This work presents a tool that provides support for software engineer in elaboration of a requirements specification document through reuse requirements patterns.*

Link: *https://youtu.be/GmAKKeX1IKQ*

Resumo. *Estudos apontam que problemas relacionados com a engenharia de requisitos são amplamente reconhecidos por afetar a qualidade do software e impactar a eficácia em seu processo de desenvolvimento. O reúso de conhecimentos obtidos de projetos anteriores pode facilitar a identificação e a escrita de requisitos na elaboração de um documento de requisitos completo e consistente. Padrão de software é uma solução para captar e reutilizar conhecimento de diferentes contextos no desenvolvimento de software. Neste trabalho é apresentado uma ferramenta que fornece apoio ao engenheiro de software na elaboração de um documento de especificação de requisitos por meio do reúso de padrões de requisitos.*

1. Introdução

Contextualização: Ketabchi, Sani e Liu (2011) afirmam que o reúso auxilia para melhorar a qualidade do software e minimizar o tempo e custos gastos em seu desenvolvimento. Além disso, o reúso propicia o aproveitamento de conhecimentos adquiridos em projetos anteriores, o que possibilita maior sucesso na elaboração de novos projetos [Palomares et al. 2013].

Uma forma de reutilizar conhecimento em diversos contextos no desenvolvimento de software é com o uso de padrões de software. Neste trabalho são utilizados padrões de requisitos, previamente elaborados pelos autores, para auxiliar na especificação do Documento de Requisitos (DR) de um software [Barcelos 2016].

Um Padrão de Requisito de Software (PRS) corresponde a um artefato que fornece informações de forma que possam ser reusados em contextos e problemas bem definidos para a completa e consistente elaboração de um DR.

Um padrão é apresentado por meio de uma estrutura, sendo recomendados quatro elementos como essenciais: nome do padrão (descrição geral da aplicabilidade), problema (o que pretende resolver), solução (descrição de como obter o resultado desejado) e consequências (implicações com o uso do padrão) [Gamma et al. 1995].

Motivação: A partir da dificuldade que engenheiros de software têm para a especificação dos requisitos, este artigo apresenta uma ferramenta para auxiliá-los na escrita de DR utilizando de padrões de requisitos. Essa ferramenta tem por base um conjunto de padrões de requisitos que atendem ao domínio de Sistema de Informação (SI) desenvolvidos por Barcelos (2016).

No desenvolvimento de software existem requisitos que são de natureza similar ou que aparecem com frequência na maioria dos softwares, o que indica um possível padrão [Withall 2007]. Essa similaridade pode ser encontrada em sistemas de diversos domínios e referem-se, geralmente, aos requisitos funcionais que descrevem a entrada e armazenamento de dados. Em SI, por exemplo, além desses, é comum encontrar requisitos funcionais para o processamento de transações, relatórios ou consultas de informações gerenciais e regras de negócio. Assim, padrões de requisitos que atendam a esses requisitos podem proporcionar ao engenheiro de software a reutilização de experiências bem-sucedidas, facilitando a identificação e a escrita desses durante a elaboração de um DR. Dessa forma, esse documento será mais completo e consistente, aumentando a qualidade do sistema a ser desenvolvido.

O conjunto de padrões elaborado auxilia o desenvolvedor para que detalhes importantes, como requisitos relacionados a outros, não sejam esquecidos ou omitidos. Os padrões existentes referem-se às operações usuais e algumas regras de negócio comumente existentes em SI. Assim, não há uma estratégia definida para uso dos padrões, pois eles são necessários a todo desenvolvimento de um SI.

Os padrões de requisitos podem ser usados tanto na identificação como também na escrita dos requisitos de um DR. Dessa forma, infere-se que pode haver redução da carga de trabalho, pois existe um roteiro a ser seguido, o que facilita a comunicação entre o desenvolvedor e os *stakeholders*.

Problema: A qualidade do software está diretamente ligada à especificação de requisitos. Estima-se que descobrir e corrigir um problema após a entrega do software, pode ser cem vezes mais caro do que se essa correção ocorrer durante as fases iniciais do desenvolvimento [Boehm and Basili 2001]. Nesse sentido, há relatos de que a completa compreensão e especificação de requisitos está entre as tarefas mais difíceis enfrentadas por um engenheiro de software [Pressman, 2011].

Este trabalho tem por objetivo mostrar uma ferramenta que apoia as tarefas de especificação de um software por meio de padrões de requisitos, bem como a instanciação desses padrões para a elaboração de um DR.

Contribuições: Essa ferramenta efetivamente possibilita que o engenheiro de software, durante a elaboração do DR, especifique os requisitos necessários às operações básicas de um SI; estabeleça o relacionamento entre os padrões; tenha a sugestão de alguns detalhes específicos da solução, como por exemplo os atributos necessários em

um cadastro; reutilize requisitos especificados em projetos concluídos com ou sem alteração e observe a definição de relacionamento de dependência entre os requisitos instanciados.

Organização do trabalho: Este trabalho está dividido em três seções, além dessa de introdução. Na Seção 2 são apresentados os conceitos sobre padrões de requisitos. Na Seção 3 a ferramenta proposta é exibida parcialmente por meio de exemplos. Na Seção 4 são comentadas as considerações finais e sugestões de trabalhos futuros.

2. Padrões de Requisitos

Palomares et al. (2013) criaram um conjunto de padrões para representar os requisitos não funcionais que podem ser utilizados em diversos domínios. Ao contrário disso, a reutilização dos requisitos funcionais na maioria das vezes só é possível para um determinado domínio de software.

Withall (2007) apresenta um catálogo com trinta e sete PRS em oito domínios, que favorecem a escrita de requisitos, atendendo às funções específicas de um domínio.

Para a elaboração da ferramenta aqui apresentada foi estabelecida uma estrutura de apresentação para a especificação de padrões (Quadro 2.1) com base nos padrões do Gamma et al. (1995) e de Withall (2007). Alguns elementos foram adicionados como domínio, tipo e padrões relacionados.

Quadro 2.1 – Estrutura Adotada para Apresentação dos Padrões

Elemento do Padrão	Descrição
Nome	Deve ser único e refletir a aplicabilidade do padrão.
Domínio	Corresponde ao domínio de aplicação do padrão.
Propósito	Descreve o objetivo da aplicação do padrão.
Problema	Descreve a situação em que o padrão pode ser aplicado.
Consequência	Descreve as consequências de se utilizar o padrão.
Tipo	Pode ser: Funcional, não funcional ou regra de negócio.
Solução	Apresenta um *template* para a escrita da parte fixa e variável do requisito que o padrão deve representar. A notação <...> é usada para descrever a parte variável que é denominada de parâmetro e deve ser substituída pelos dados pertinentes ao requisito .
Padrões Relacionados	Especifica os padrões relacionados, complementares ao padrão em questão. Possibilita a indicação de outros possíveis padrões que podem ser usados com este.

3. Ferramenta para a Elaboração de Documento de Requisitos com Padrões

Palomares et al. (2014) afirmam que há dificuldade de utilizar padrões sem um apoio computacional, pois o desenvolvedor pode não conhecer todas as particularidades envolvidas para a especificação de um software. Considerando as observações feitas por Palomares et al. e a usabilidade dos padrões, foi desenvolvida uma ferramenta para auxiliar engenheiros de software na atividade de elicitação de requisitos.

A ferramenta está disponível para baixar no endereço http://advanse.dc.ufscar.br/index.php/tools e a sua utilização é livre.

Na Seção 3.1 é abordada a construção da ferramenta. Na Seção 3.2 são descritas as funções da ferramenta.

3.1. Construção da Ferramenta

A construção da ferramenta seguiu o modelo de processo iterativo e incremental, o que permitiu melhorar a sua funcionalidade, usabilidade e a inclusão de novas ideias.

A linguagem Java foi escolhida juntamente com a plataforma Java SE (*Standard Edition*), com o banco de dados MySQL *Server Community*, a biblioteca JDBC (*Java Database Connectivity*) do MySQL para permitir a conexão com o banco de dados e a biblioteca iText para a criação do DR no formato *PDF*.

A ferramenta possui três módulos: a) o de especificação e gestão dos padrões, que geralmente fica sob a responsabilidade de um engenheiro de software com mais experiência; b) o de instanciação dos padrões durante a elicitação de requisitos; e c) o de funções básicas de manutenção.

Na Figura 3.1 o modelo conceitual da ferramenta é apresentado utilizando um modelo de classes UML.

Para a escrita de um requisito (*Requisito*), deve existir um projeto (*Projeto*), que requer a seleção do cliente (*Cliente*), do usuário (*Usuário*) responsável pela especificação do DR e do subdomínio (*Subdomínio*) que orienta o engenheiro de software no reúso dos requisitos especificados em projetos futuros. O *template* da solução do padrão (*Padrão*) é fornecido para a especificação do requisito (*Requisito*), que possui uma parte do texto fixa e outra variável. A parte variável é chamada de parâmetro do requisito (*Parâmetro Requisito*) que foi fornecida pelo padrão para receber os valores informados (*Valor Parâmetro Requisito*) pelo engenheiro de software.

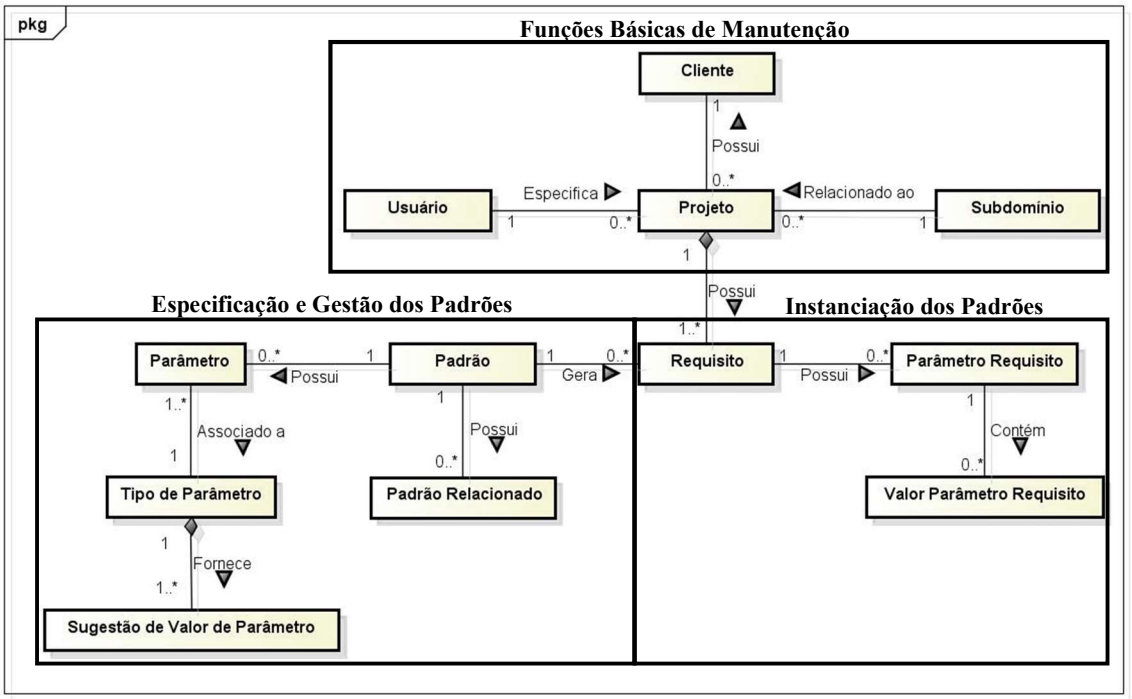


Figura 3.1 – Modelo Conceitual da Ferramenta

Os requisitos gerados em projetos concluídos ficam armazenados em um repositório que permite a reutilização de requisitos em projetos futuros, com a possibilidade de alteração desses requisitos, se necessário.

3.2. Exemplificando a escrita de um DR por Meio da Instanciação de Padrões

A especificação de requisitos de um sistema de Pedido de Venda será utilizada para exemplificar a ferramenta. A Visão Geral do sistema é exibida no Quadro 3.2.1.

Quadro 3.2.1 – Visão Geral do Sistema de Pedido

Os clientes devem estar cadastrados para a emissão dos pedidos. O sistema somente deve permitir a realização de pedidos a prazo apenas para os clientes que tiverem limites de crédito disponível. O sistema deve permitir emitir relatório de pedidos realizados na data atual.

A elaboração do DR por meio da ferramenta inicia-se com o cadastro do projeto, sendo necessário fornecer: um nome para o projeto, a visão geral (fará parte do DR), a seleção do cliente e analista responsáveis (previamente cadastrados), a data de início do projeto, o status (inicialmente deve ser "aberto" para indicar que o projeto está em andamento) e a seleção do subdomínio, para orientar o engenheiro de software na reutilização futura dos requisitos desse projeto em outros.

Após o cadastro do projeto deve-se selecionar os padrões de acordo com as necessidades declaradas pelos *stakeholders*. As informações sobre o padrão selecionado são apresentadas na aba "*Informações Básicas*" como pode ser visto na Figura 3.2.1.

De acordo com o colocado no Quadro 3.2.1 é necessário armazenar os clientes em banco de dados. Logo, o padrão *Incluir Informação* deve ser instanciado utilizando-se o botão [*Instanciar*].

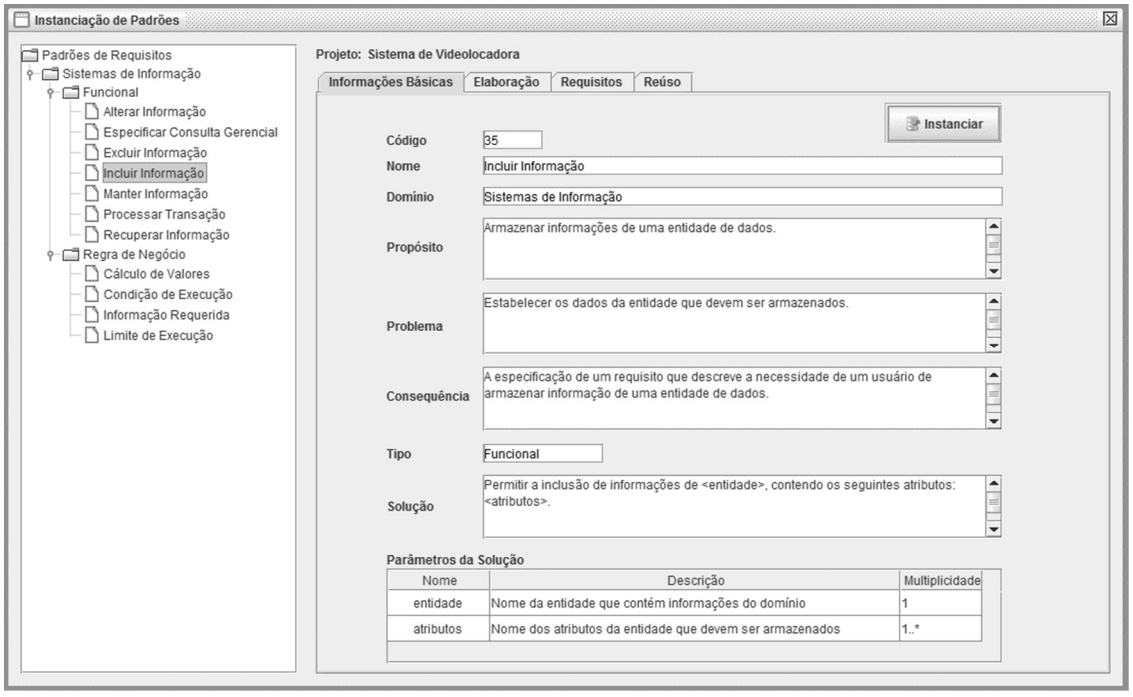


Figura 3.2.1 – Instanciação do Padrão Incluir Informação

Na Figura 3.2.2 é exibida a aba "*Elaboração*", usada para a escrita de um requisito para a inclusão de cliente. Para tanto deve-se: a) definir de uma *Descrição* para o requisito, para facilitar a sua localização; b) no *Template* é apresentado o texto da solução com os parâmetros (b1), que devem ser preenchidos com as informações fornecidas pelos *stakeholders*; c) após ter o parâmetro selecionado há duas opções: a escrita manual dos dados correspondentes e clicar no botão [*Novo*] (c1) ou então a seleção dos dados a partir das sugestões fornecidas pelo repositório da ferramenta (c2); d) com a seleção do botão [*Visualizar*] ocorre a substituição do parâmetro no *template* pelos dados atribuídos ao parâmetro. Esse passo não é obrigatório sendo que a substituição ocorrerá quando o botão [*Salvar*] for pressionado para concluir a escrita do requisito; e) finalmente, pressionar o botão [*Salvar*] para concluir a escrita do requisito.

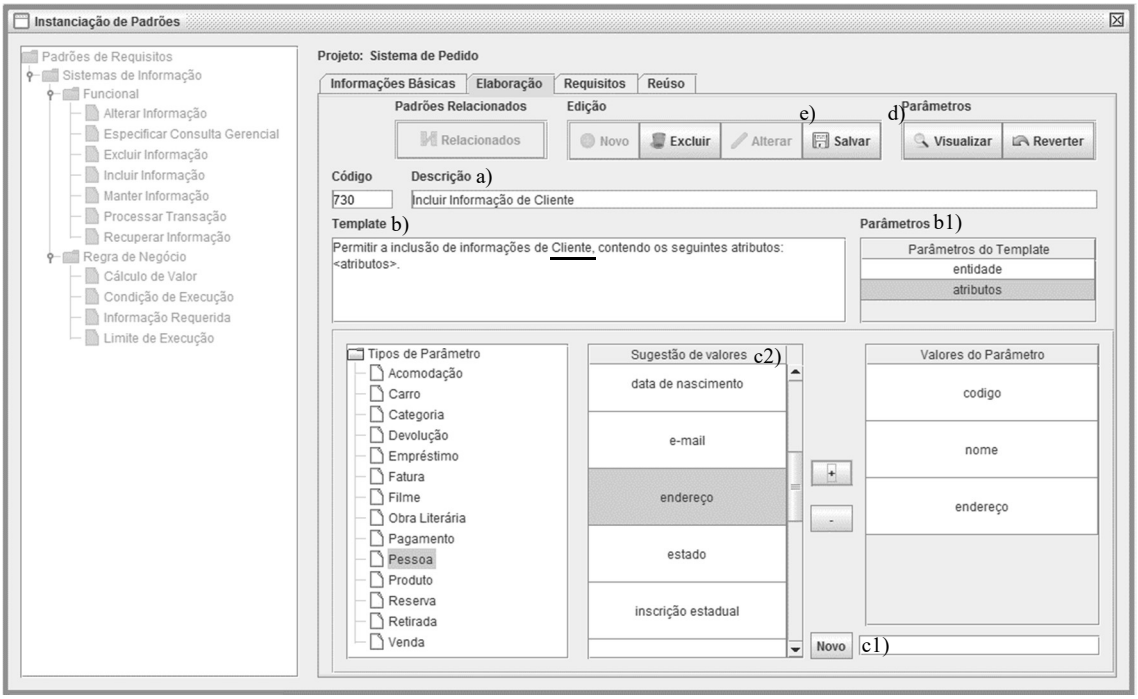


Figura 3.2.2 – Escrita do Requisito para Cadastro de Cliente

Um outro requisito do sistema exemplo é o processamento do pedido de venda, que deve ser feito com o padrão "*Processar Transação"*. Sua instanciação ocorre da mesma forma que o padrão "*Incluir Informação*"*,* comentado anteriormente.

Ao salvar um requisito a ferramenta pode sugerir ao engenheiro de software o uso de algum padrão relacionado ao instanciado, se existir, para complementar a especificação do requisito. Essa sugestão é realizada por meio do botão [*Relacionados*] que apresenta um número que corresponde à quantidade de padrões relacionados. Na Figura 3.2.3 pode-se observar esse caso. A instanciação desses padrões também ocorre na aba "*Elaboração*".

Para atender ao requisito de permitir a realização de pedidos a prazo apenas para os clientes que tiverem limites de crédito disponível, o padrão *Processar Transação* possui o padrão relacionado *Condição de Execução* que deve ser utilizado para essa finalidade. Finalmente para atender ao requisito de emitir um relatório, o padrão *Especificar Consulta Gerencial* deve ser instanciado.

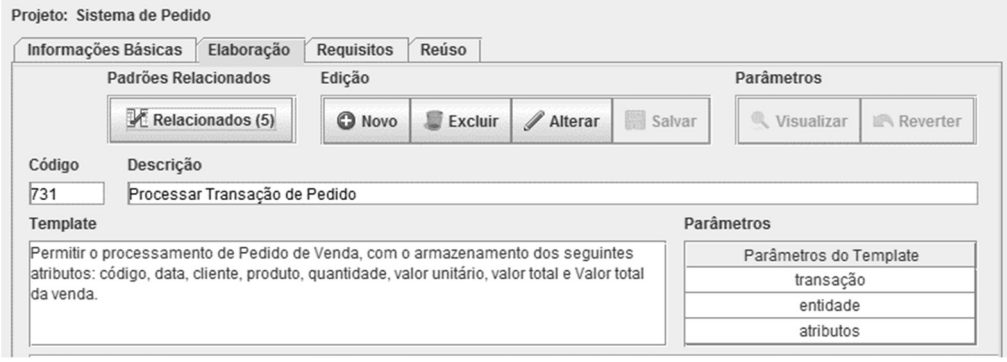


Figura 3.2.3 – Escrita do Requisito para Cadastro de Cliente

No módulo de Instanciação de Padrões, a aba "*Requisitos*" (Figura 3.2.4) tem por função apresentar os seguintes itens: a) todos os requisitos do projeto, ao selecionar um requisito também são apresentados os requisitos relacionados, caso existam; b) permitir a seleção do requisito para a edição, visualização ou exclusão e fornecer sugestões de padrões relacionados ao requisito selecionado; c) permitir o estabelecimento de relacionamento (rastreabilidade) entre os requisitos; d) permitir a geração do documento com todos os requisitos instanciados, no formato *PDF*. Na Figura 3.2.5 é exibido parte do DR gerado pela ferramenta para o sistema exemplo.

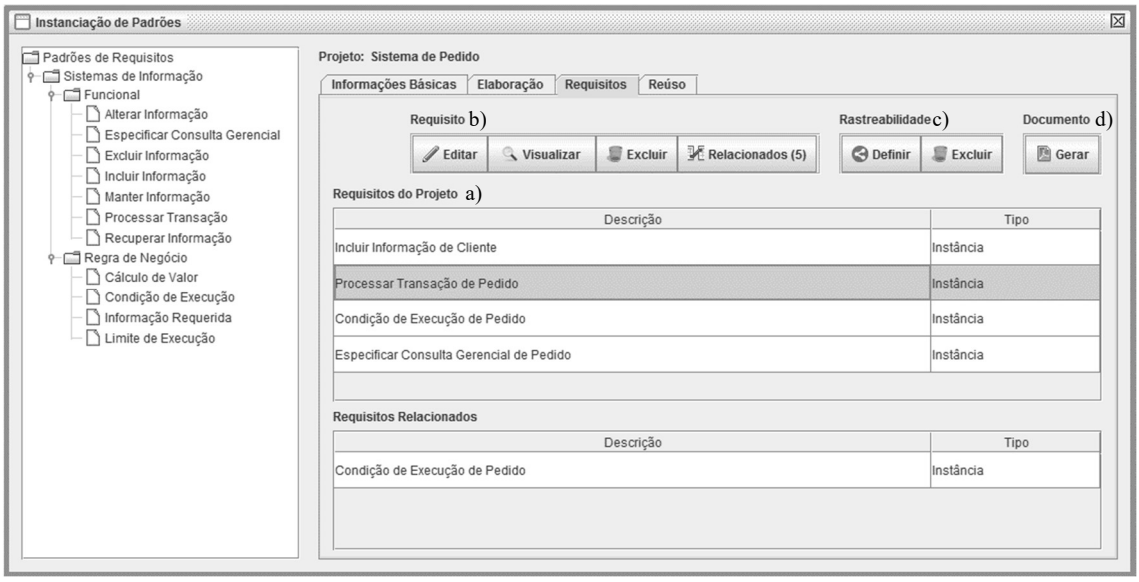


Figura 3.2.4 – Documento de Requisitos de Exemplo

Identificador:	Descrição	Depende de:
RF1	Permitir a inclusão de informações de Cliente, contendo os seguintes atributos: codigo, nome, endereço, cidade, estado, cpf, RG, telefone 1 e telefone 2.	
RF2	Permitir o processamento de Pedido de Venda, com o armazenamento dos seguintes atributos: código, data, cliente, produto, quantidade, valor unitário, valor total e Valor total da venda. RN2.1. A operação de pedido somente deve ser permitida se o cliente possuir limite de crédito disponível.	
RF3	Permitir a impressão de um relatório de pedidos realizados na data atual, contendo os seguintes atributos: código do cliente, nome do cliente, cpf e valor total do pedido.	

Figura 3.2.5 – Documento de Requisitos de Exemplo

A aba "*Reúso* apresenta os requisitos do repositório que foram instanciados em outros projetos, referentes ao padrão selecionado. Esses requisitos podem ser reusados no projeto atual integralmente ou com a possibilidade de ter alterações.

4. Considerações Finais

Este trabalho apresentou uma ferramenta para apoiar ao engenheiro de software na especificação de DR usando padrões de requisitos.

A avaliação dessa ferramenta foi realizada por meio de estudos de casos com estudantes de graduação e pós-graduação, que indicou que o DR elaborado é mais completo, um aumento na produtividade e que a ferramenta é de fácil utilização. Os estudantes com menos experiência se beneficiaram com o uso da ferramenta principalmente quanto à colocação de padrões relacionados aos requisitos funcionais que possuíam regras de negócio específicas. Os estudantes mais experientes avaliaram que a ferramenta proporcionou a elaboração de um DR com maior qualidade, uma vez que os padrões de requisitos conduzem o engenheiro de software para que detalhes não sejam esquecidos ou para que sejam especificados de forma mais completa.

A usabilidade da ferramenta também foi considerada pelos estudantes, uma vez que ela apresenta características comuns existentes em outras ferramentas de software de uso constante por eles e pela comunidade.

Como trabalho futuro é sugerido o uso mais intenso da ferramenta a fim de identificar possíveis melhorias quanto à funcionalidade e usabilidade e a implementação na ferramenta de outras seções para o DR, com a possibilidade de descrever outras informações sobre o sistema, usando como referência o *template* fornecido pela norma IEEE *Std* 830.

Referências

Barcelos, L. (2016) "Especificação de Requisitos no Domínio de Sistemas de Informação Com o Uso de Padrões". São Carlos/SP: Universidade Federal de São Carlos.

Boehm, B. and Basili, V. R. (2001) "Software Defect Reduction Top 10 List". IEEE Computer Society.

Gamma E, Helm. R, Johnson, R. and Vlissides, J. (1995) "Padrões de Projeto". Porto Alegre/RS: Artmed.

Ketabchi, S.; Sani, N. K. and Liu, K. (2011) "A Norm-Based Approach towards Requirements Patterns". IEEE Annual Computer Software and Applications Conference.

Palomares, C., Franch, X., and Quer, C. (2014) "Requirements Reuse and Patterns: A Survey. In Requirements Engineering". Foundation for Software Quality. Switzerland: Springer.

Palomares, C., Quer, C., Franch, X., Renault, S. and Guerlain, C. (2013) "A Catalogue of Functional Software Requirement Patterns for the Domain of Content Management Systems". ACM Symposium on Applied Computing.

Pressman, R. S. (2011) "Engenharia de Software - Uma Abordagem Profissional". 7ª. ed. Porto Alegre/RS: McGraw-Hill.

Withall, S. (2007) "Software Requirement Patterns". Redmond, Washington: Microsoft Press.

Model2gether: a tool to support cooperative modeling involving blind people

Leandro Luque[1,2], Christoffer L. F. Santos[1], Davi O. Cruz[2], Leônidas O. Brandão[3], Anarosa A. F. Brandão[1]

[1]Escola Politécnica - University of Sao Paulo (USP)
158 Prof. Luciano Gualberto Avenue – 05.508-010 – Sao Paulo – SP – Brazil

[2]Department of Systems Analysis and Development
Sao Paulo State Technological College (Fatec) – Mogi das Cruzes, SP – Brazil

[3]Institute of Mathematics and Statistics – University of Sao Paulo (USP)
Sao Paulo, SP – Brazil

leandro.luque@usp.br, leo@ime.usp.br, anarosa.brandao@usp.br

Abstract. *Some models, such as those of UML, data flow diagrams, and entity-relationship diagrams have a strong dependence on their graphical representations. The frequent use of these models in the computing field creates obstacles to the inclusion of blind people in industry and academia. These obstacles are related not only to individual access and editing of such models, as well as to scenarios in which other sighted and blind people work cooperatively. In this paper, we present Model2gether, a web-based tool that support the inclusion of blind people in cooperative modeling. We have not found any equivalent tool in terms of awareness and communication mechanisms - essential features to cooperative activities in academia and industry.*
Link to video:https://www.youtube.com/watch?v=2OrkHJ8F7uw

1. Introduction

Models and modeling play an important role in the computing field as a means to understand, design and document various aspects of hardware and software systems [1, 2, 3]. Many models used in this field have a tight dependence on graphical representations, often in form of diagrams. Examples of such models are Data Flow Diagrams (DFD) [4], Entity-Relationship Diagrams (ERD) [5], and the Unified Modeling Language (UML) diagrams [6].

The use of such models, called hereafter graphical models, creates obstacles to the inclusion of blind people in computing-related courses and industry [7, 8, 9, 10]. It happens, among other reasons, because the tools blind people use when interacting with computers (e.g. screen readers) cannot interpret the semantic of pixel sets. Therefore, they do not recognize the relevant content of graphical model images.

Despite the extensive literature on the accessibility of these models considering individual activities [11, 12, 13, 14, 15, 16, 17, 18], many activities conducted in academia and industry have a cooperative nature. Also, the existence of solutions for individual activities does not guarantee the possibility of performing cooperative activities since there are several features of the latter that are not present in the former [19]. Literature

on cooperative modeling involving blind people is scarce [20] and we have found only one software tool that partially support such modeling activities [21, 20]. In this context, we developed an web-based tool to fill this gap. This tool is called Model2gether and is available at http://www.model2gether.com:6223.

2. Model2gether

In this section, we describe the requirements and architecture of Model2gether. It is a free software - distributed under the GNU General Public License (GPL) - that supports cooperative modeling involving both blind and sighted people. It currently allows cooperative modeling of UML use case models, but we have been working on extensions to Class and Entity-Relationship models.

3. Requirements

The requirements for a tool that supports cooperative modeling involving blind people must consider both accessibility and cooperative concerns. Accessibility requirements must assure that a model can be accessed and edited by users. Cooperative requirements involve three types of social mechanisms [19]:

- Conversational mechanisms: to facilitate the flow of speech and to help overcoming breakdowns during it;
- Coordination mechanisms: to allow people to work and interact together;
- Awareness mechanisms: to find out what is happening, what others are doing and, conversely, to let others know what you are doing.

In previous works [22, 23], we defined a set of requirements for e-learning activities involving blind people. Despite they were defined focusing on an educational setting, as they consider all the aforementioned concerns, it is expected that cooperative activities in industry are covered as well. All these requirements (see Table 1) are implemented in Model2gether.

The tool implements two different interfaces: a screen-reader compatible interface for blind users (Figure 2) and a graphical interface for sighted users (Figure 3). In the former, models are specified through a DSL - Domain Specific Language [24]. The elements that may be modeled through the DSL are: actors, use cases, associations, dependencies (inclusion and extension), as well as actor and use case inheritance. Interaction is possible through shortcuts - a combination of modifiers and keys (e.g. Ctrl+Shift+A) - and keyboard navigation. In this interface, there are options to control awareness and communication mechanisms. If the user checks ”Follow model updates”, he/she receives text messages (e.g.: A new use case was created: Borrow Material) every time another user changes the model (visual changes are not reflected to blind people). Nevertheless, if the user checks ”Listen to beeps when changes are made to the model”, beeps are produced when another user changes the model. Another possibility is to perceive changes in the textual model representation, but not be warned about these changes (”Allow real-time changes on the textual model representation”). Additionally, there are shortcuts to navigate through the history of actions other users had done. This history is updated independently of the aforementioned options.

The interface for sighted users shows use case models in their diagrammatic format and interaction is possible through mouse and keyboard. A model may be shared

As a sighted participant, I want to	1: create and edit diagrams using the graphical UML notation 2: have access to diagrams created by sighted participants through the graphical UML notation 3: have access to diagrams created by blind participants through the graphical UML notation	Accessibility
	4: follow changes in diagram editing activities through the graphical UML notation in order to understand and collaborate	Awareness
	5: follow the highlights made by other participant during activities and navigate throughout the highlighted elements using the graphical UML notation, to be able to follow the explanation and collaborate	Communication and Accessibility
As a blind participant, I want to	6: create and edit accessible diagrams 7: access the diagrams created by the sighted participants 8: access the diagrams created by the blind participants	Accessibility
	9: to follow changes in diagram editing activities in order to understand and collaborate	Awareness
	10: follow the highlights made by other participant during activities and navigate throughout the highlighted elements in order to be able to follow the explanation and collaborate	Communication and Accessibility
As a participant, I want to	11: highlight elements of the diagrams I am presenting, creating or editing, so other participants can follow the explanations and collaborate	Communication

Figure 1. Requirements to include blind people in e-learning activities that use graphical models, adapted from [22]

with other participants through a sharing option, shown in the main screen, in which all use case models owned by the user or shared with him/her are shown (Figure 4). Finally, there are options for a user to manage his/her profile.

Model2gether's accessibility was checked through its conformance with WCAG 2.0. To do that, we used an automatic free service available at achecker.ca. No known problems nor likely problems were found by the service.

4. Software Architecture

The tool is organized in 4 logical modules, as shown in Figure 5. The domain module contains Plain Old Java Objects (POJO) that implements domain abstractions, such as User and Sharing. Additionally, it contains a meta-model that can represent graph-based (with graphs nested in nodes) models. Figure 6 shows the meta-model class diagram. All meta-model elements implements the Observer design pattern. Doing so, any change to the meta-model is immediately notified to its observers.

The web-independent services are located in the Generic Service Module. In addition to other services, it manages the persistence of classes in the domain module through Data Access Objects (DAO) and a JPA-based implementation. Additionally, it contains classes responsible for producing changes in the meta-model and for managing the set of rules that must be followed to produce a valid model instance. Both Factory Method and Strategy design patterns are used to do that. All services in this module are accessible through a Facade, that simplifies its usage.

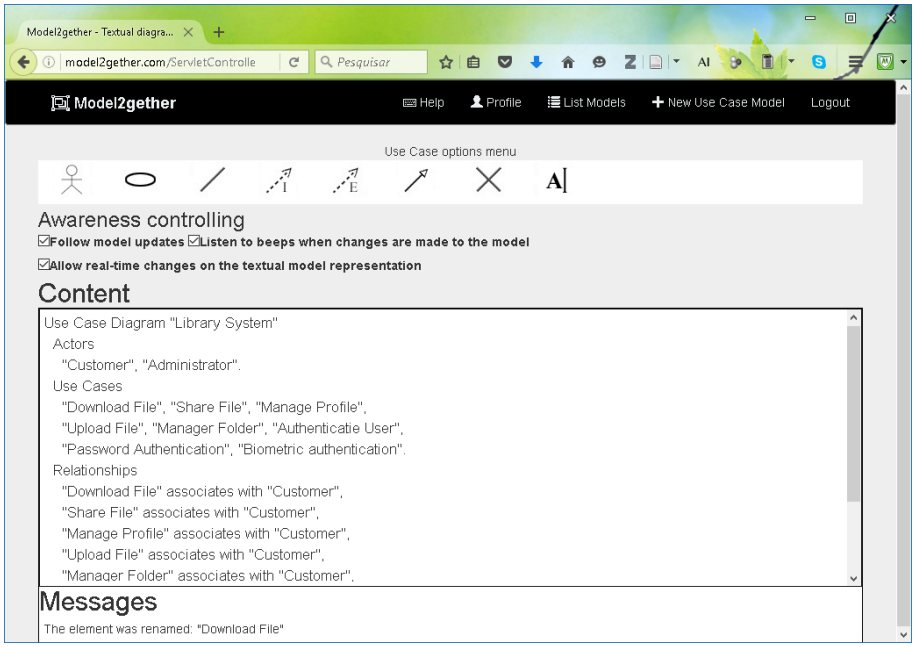


Figure 2. Model2gether interface for blind users.

The Web Service Module is also responsible for receiving application requests and controlling the application flow. It is implemented through Spring MVC controllers and view-models. Excluding the sign in, sign up, and error pages, all other resources are filtered through an intercepting filter.

The Presentation module, by its time, is responsible for presenting and controlling interactions with the user interface. It was written using JavaServer Pages (JSP), HTML 5, CSS 3, and Javascript. In this module, a Javascript instance of the meta-model is used to represent the model being edited. One of the meta-model observers is a Javascript controller that sends messages to the Web Service Module through a websocket. Additionally, this code is responsible for receiving messages from the websocket and requesting changes in the textual or graphical model representation. Currently, the tool uses the JSUML2 library to draw use case diagrams. However, we are working on a new Javascript-based UML library with improved usability.

5. Case Study

In April 2016, we carried out a user study with 4 blind and 1 sighted people. The blind participants were invited from the Brazilian Blind Programmers List. The sighted participant was one of the authors.

Firstly, we invited the blind participants to test the tool in order to assure its accessibility. For this, they were asked to conduct the following activities: (i) open the tool website and create an account; (ii) sign-in into the system; (iii) localize the help menu, open and read it; (iv) open a use case model that was shared with them and identify the actors, use cases, and relationships; (v) create a new use case model based on a textual description; and finally (vi) share this diagram with the authors.

Before the tests, a lecture on use case modeling was provided by the sighted participant. Participants were allowed to make questions at any time. The lecture was used to

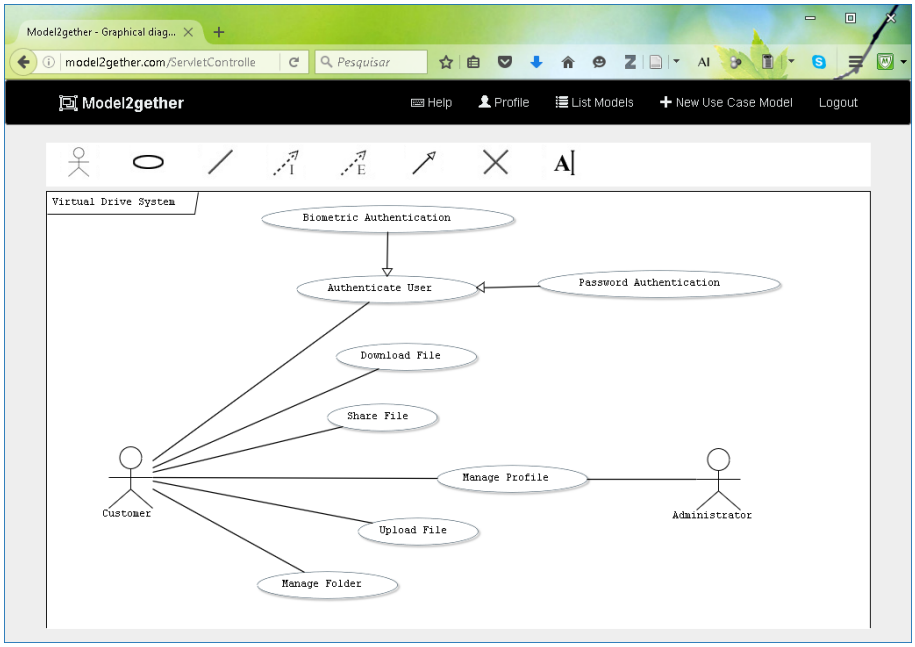


Figure 3. Model2gether interface for sighted users.

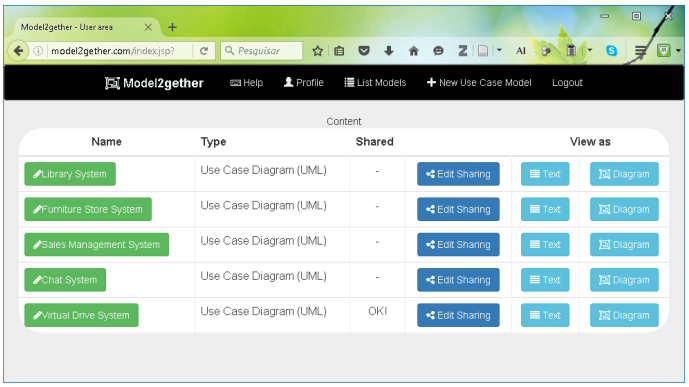


Figure 4. Model2gether main menu.

assure that all participants shared the same definition about concepts that would influence cooperative modeling.

Therefore, each blind user worked in pairs with the sighted participant in order to cooperatively model a system. The blind participant received a system description and modeled together with the sighted participant. Different awareness and communication strategies were adopted - all possible state combinations of awareness and communication checkboxes. All blind participants were very excited with the tool and gave positive feedback (e.g. ”It was the first time I felt comfortable following a class that contains graphical content”). One of them used the tool a week later to work together with colleagues in order to create a use case model for a course he was taking.

6. Comparison with Similar Tools

To date, there are two groups of tools with which Model2gether is related to. The ones that allow the specification of UML models through text (MetaUML, PlantTextUML Ed-

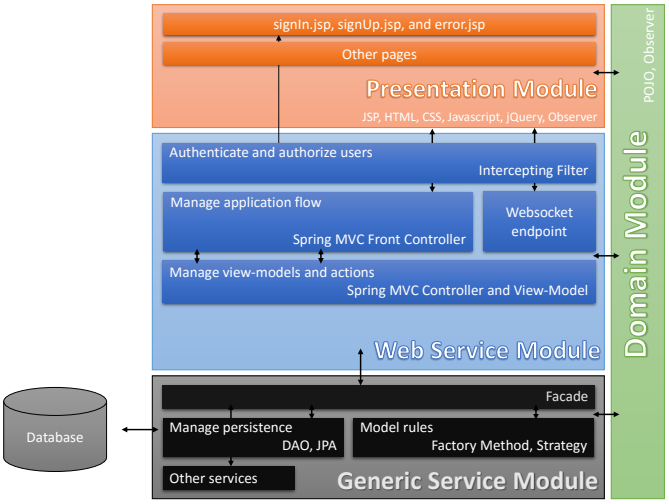


Figure 5. Model2gether architecture.

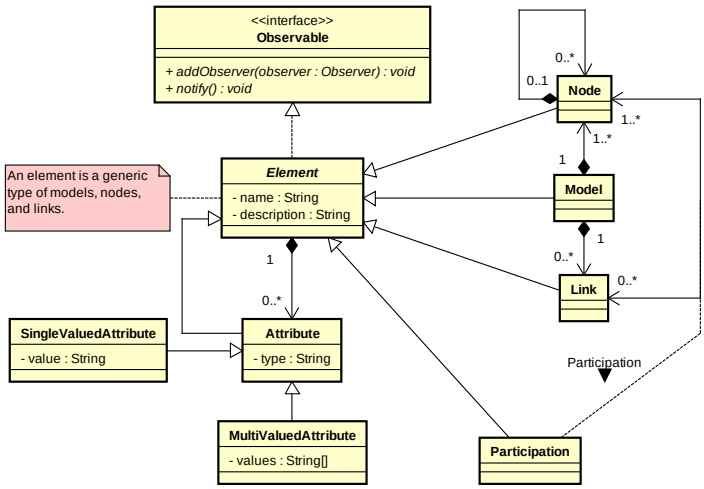


Figure 6. Model2gether meta-model.

itor, PlantUML, UMLetino, and yUML) and the ones that seek to support cooperative modeling involving blind people, as CCMi and AWMo.

A comparison of Model2gether with the ones listed above is provided in Table 1.

7. Conclusion

In this paper, we presented Model2gether, a free tool that aims at supporting cooperative modeling involving blind people. Given the importance of such activities in academia and industry, as well as the lack of solutions that implement requirements to support real-time cooperative modeling, Model2gether may be considered a promising tool. It may be seen as a means to reduce the obstacles faced by blind people in computing-related courses and in industry.

Usability tests performed with the tool indicate that it contributes to lectures involving blind people, as well as to group modeling activities. Still, its flexible architecture

Table 1. Tool's comparison

	Cooperative Modeling			Textual UML Editors				
Feature	Model2gether	CCMi	AWMo	MetaUML	PlantTextUML Editor	PlantUML	UMLetino	yUML
1. Allow textual editing	✓		✓	✓	✓	✓	✓	✓
2. Allow graphical editing	✓	✓	✓				✓	
3. Support model sharing	✓	✓	✓	✓	✓	✓	✓	✓
4. Support model storing	✓	✓		✓	✓	✓	✓	✓
5. Allow exporting to image	✓	✓			✓	✓	✓	✓
6. Support Real-Time (RT) cooperation	✓	✓						
7. Implements RT communication mechanisms	✓							
8. Implements RT awareness mechanisms	✓							
9. Implements RT coordination mechanisms		✓						

allows easily addition of new models.

As future work, in addition to implementing new models (e.g. UML class model and entity-relationship model), we plan to implement new features, such as support for tactile devices, DSL customizations, and coordination mechanisms. Furthermore, controlled experiments with more users will be conducted.

References

[1] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[2] Hassan Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

[3] Jeremiah Hayes. *Modeling and analysis of computer communications networks*. Springer Science & Business Media, 2013.

[4] Qing Li and Yu-Liu Chen. *Modeling and Analysis of Enterprise and Information Systems: from requirements to realization*. Springer Publishing Company, Incorporated, 2009.

[5] Peter Pin-Shan Chen. The entity–relationship model: toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[6] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[7] Richard Connelly. Lessons and tools from teaching a blind student. *Journal of Computing Sciences in Colleges*, 25(6):34–39, 2010.

[8] Karin Müller. *How to make unified modeling language diagrams accessible for blind students*. Springer, 2012.

[9] L. Luque, E. S. Veriscimo, G. C. Pereira, and L. V. L. Filgueiras. Can We Work Together? On the Inclusion of Blind People in UML Model-Based Tasks. In P. M. Langdon, J. Lazar, A. Heylighen, and H. Dong, editors, *Inclusive Designing*, pages 223–233. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-05095-9_20.

[10] Sarah Coburn and Charles B Owen. UML diagrams for blind programmers. In *Proc. of the 2014 ASEE North Central Section Conference. Oakland, USA*, pages 1–7, 2014.

[11] Matt Calder, Robert F Cohen, Jessica Lanzoni, Neal Landry, and Joelle Skaff. Teaching data structures to students who are blind. *ACM SIGCSE Bulletin*, 39(3):87–90, 2007.

[12] Filipe Del Nero Grillo and Renata Pontin de Mattos Fortes. Accessible modeling on the web: a case study. *Procedia Computer Science*, 27:460–470, 2014.

[13] Filipe Del Nero Grillo, Renata Pontin de Mattos Fortes, and Daniel Lucrédio. Towards collaboration between sighted and visually impaired developers in the context of model-driven engineering, 2014.

[14] Rubén Iglesias, Sara Casado, T Gutierrez, JI Barbero, CA Avizzano, S Marcheschi, and M Bergamasco. Computer graphics access for blind people through a haptic and audio virtual environment. In *Haptic, Audio and Visual Environments and Their Applications, 2004. HAVE 2004. Proc. of the 3rd IEEE International Workshop on*, pages 13–18. IEEE, 2004.

[15] Alasdair King, Paul Blenkhorn, David Crombie, Sijo Dijkstra, Gareth Evans, and John Wood. Presenting UML Software Engineering Diagrams to Blind People. In Klaus Miesenberger, Joachim Klaus, Wolfgang L. Zagler, and Dominique Burger, editors, *Computers Helping People with Special Needs*, number 3118 in Lecture Notes in Computer Science, pages 522–529. Springer Berlin Heidelberg, July 2004. DOI: 10.1007/978-3-540-27817-7_76.

[16] Claudia Loitsch and Gerhard Weber. *Viable haptic UML for blind people*. Springer, 2012.

[17] Oussama Metatla, Nick Bryan-Kinns, and Tony Stockman. Constructing relational diagrams in audio: the multiple perspective hierarchical approach. In *Proc. of the 10th international ACM SIGACCESS conference on computers and accessibility*, pages 97–104. ACM, 2008.

[18] Luciano TE Pansanato, Christiane E Silva, and Luzia Rodrigues. Uma experiência de inclusão de estudante cego na educação superior em computação. In *XX Workshop on Computing Education*, 2012.

[19] Jenny Preece, Helen Sharp, and Yvonne Rogers. *Interaction Design-beyond human-computer interaction*. John Wiley & Sons, 2015.

[20] Oussama Metatla, Nick Bryan-Kinns, Tony Stockman, and Fiore Martin. Cross-modal collaborative interaction between visually-impaired and sighted users in the workplace. In *Proc. of the Designing Interactive Systems Conference 2012*. Georgia Institute of Technology, 2012.

[21] Oussama Metatla, Nick Bryan-Kinns, Tony Stockman, and Fiore Martin. Designing for collaborative cross-modal interaction. In *Proc. of Digital Engagement 2011 the 2nd RCUK Digital Economy All Hands Meeting*, 2011.

[22] Leandro Luque, Leônidas de Oliveira Brandão, Romero Tori, and Anarosa Alves Franco Brandão. On the inclusion of blind people in uml e-learning activities. *Brazilian Journal of Informatics in Education*, 23(02):18, 2015.

[23] Leandro Luque, Leônidas O Brandão, Romero Tori, and Anarosa AF Brandão. Are you seeing this? what is available and how can we include blind students in virtual uml learning activities. In *Brazilian Conference on Informatics in Education*, volume 25, pages 204–213, 2014.

[24] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

BTestBox: an automatic test generator for B method

David Deharbe[1,2], Diego Azevedo[1], Ernesto C. B. de Matos[1], Valério Medeiros Jr.[3]

[1] Departamento de Informática e Matemática Aplicada – DIMAp/UFRN
Natal, RN – Brazil

[2]Clearsy System Engineering
Aix-en-Provence – France

[3]Federal Institute of Education, Science and Technology of Rio Grande do Norte – IFRN
Parnamirim, RN – Brazil

david@dimap.ufrn.br, {diegooliveira,ernestocid}@ppgsc.ufrn.br,
valerio.medeiros@ifrn.edu.br

Abstract. *This paper presents BTestBox, a model-based testing tool that generates test cases for code generated from B Method specifications. BTestBox receives as input a B implementation and then generates test cases to compare the execution of the generated code and the B model. Our tool uses an animation history of the B implementation to get the expected states and check if the generated code reaches the same state with the same operations. BTestBox generates a report with the results of evaluated tests. In this paper, we show how the tool can be used to test LLVM code generated from B Method specifications and present a case study which evaluated an LLVM translator, obtaining significant results.*

BTestBox video demonstration is available at:
`https://github.com/ValerioMedeiros/BTestBox`.

1. Introduction

BTestBox is a tool developed to support the validation of code generated from B specifications. Its primary objective is to assure the correctness of B translations in other programming languages. BTestBox can be used as an extension for Atelier B—a popular IDE (Integrated Development Environment) used to model, refine, verify, and generate code for B Method specifications. Our tool receives as input a B implementation and the generated code and then creates the history of a simulation using ProB[1]. After that, the code is executed and compared with the state in this history, and a report is generated.

In the current state, BTestBox is fully automatic and capable of testing the output of a translator from B to LLVM, indicating if there is an error in the translation. Our tool was used in a case study which validated the translation from B to LLVM using the B2LLVM compiler [Medeiros Jr. 2016]. The results were significant and are presented in this paper.

This paper is organized as follows: Section 2 gives a brief introduction to the B Method and the motivation behind our work; Section 3 explains basic concepts related

[1]ProB is an animator and model-checker for B models.

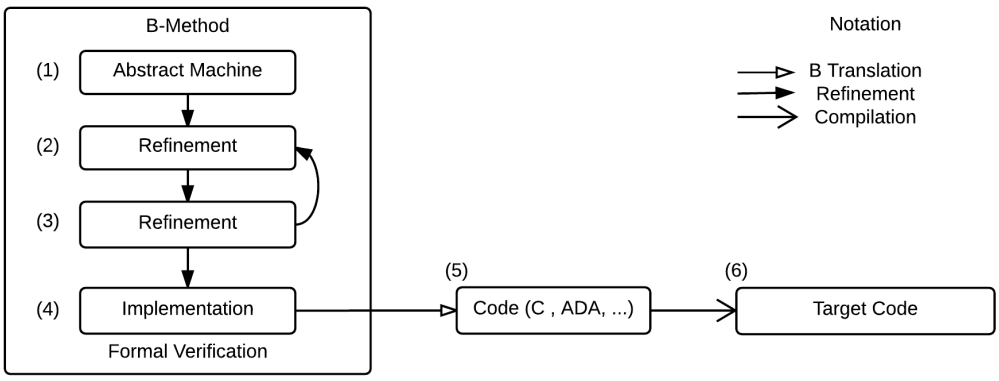


Figure 1. B-Method

to the architecture and the operational context of BTestBox; Section 4 presents an example and describes the validation process for BTestBox; Section 5 presents related work; Ultimately, Section 6 presents conclusions and the future work.

2. Motivation

The B Method is a consolidated formal method that has been used for several years in the development of critical systems [Medeiros Jr. 2016]. It is based on the abstract machine notation and on the generalized substitution theory, which was grounded over Dijkstra's notes [Dijkstra et al. 1976]. The B Method supports modular modeling; each module specifies a software component in a different abstraction level [Medeiros Jr. et al. 2009]. The main idea is to start with an abstract model of the system under development and, gradually, add details with subsequent refinements [Bjørner and Henson 2007]. Figure 1 presents an overview of this development process. The last and most concrete refinement receives the name of *algorithmic model* or *B implementation*. Its goal is to implement the model using programming language constructs. The final objective of the process is to obtain a proven implementation model [Bjørner and Henson 2007]. Such proofs ensure that the model satisfies certain properties. There is still a weak step in the development process: the production of the binary code from the B implementation (translation to a programming language and application of an off-the-shelf compiler).

Errors in compilers occur and may silently introduce bugs from correct source code [Leroy 2009]. Even if the source code meets the functional requirements, which is demonstrably the case with the B method, the object code may not meet these requirements. Indeed, eleven C compilers are identified with more than 325 compilation errors [Yang et al. 2011]. Among them, five are open source (GCC, LLVM, CIL, TCC e Open64), five are commercial and the CompCert[Leroy 2008] which offers rights to free use to non commercial purpose, but the commercial use requires a INRIA special license. Compilers and translators used in small communities have higher inherent technological risks than tools used in large scale [Stuermer et al. 2007]. In the B community, AtelierB supports several code generators. These code generators demand additional safety criteria, mainly because they are used in critical applications. In practice, a countermeasure accepted in certification standards is to exploit redundancy, by developing and applying at

least two different tool chains to produce two object programs. These programs are executed in parallel and their results are compared at run-time. When a difference is detected, the system is then put into a safe state.

Testing techniques can complement formal methods in the Verification and Validation process. These techniques can be used as an instrument for an in-depth validation of the system. Also, some certification standards require the use of software testing techniques, even for systems that already use the sound mechanisms provided by formal methods [Rushby 2008]. There are limitations to formal methods, and software testing can complement a formal verification providing tools to identify failures, exploit possible defects introduced during refinement or implementation, or during the maintenance of the code [Matos 2016].

Among the B code translators, B2LLVM is a compiler for B implementations that generates LLVM code. Actually, the creation of BTestBox has been motivated by the need to validate functionally the code produced by B2LLVM. However the scope is and it can be employed together with other code generators.

3. The Tool

3.1. Tool Architecture

Test case generation and execution using BTestBox assumes the B method has been executed, resulting in a full software component development from an abstract model to a binary implementation of this component, and consisting of the following artifacts: an abstract model, a series of refinements, the last of which being the B implementation, and the binary code derived from it by application of a code generator and a compiler. Then BTestBox provides integrated support for the following steps: 1) Generate a ProB simulation history; 2) Generate tests using the ProB [Leuschel and Butler 2003] history; 3) Execute the tests comparing the ProB history and the translated code; 4) Report the results.

BTestBox thus relies on AtelierB and ProB, two important tools to support formal development with B. Atelier-B provides features to manage B projects, including code generation and formal verification. ProB is an animator and model-checker for B models, by solving logical and mathematical expressions [Leuschel and Butler 2003]. The animation comprehends the interpretation and execution of the software's model, by execution of operations and evaluation of state transitions in the model.

The animation of B models using ProB can be done in several ways. For our tests, as default, we use random animations. Random testing selects any input from all possibilities and is commonly used to verify software reliability [Chen et al. 2013]. Random testing can effectively evaluate the software under many situations [Menzies and Cukic 2000].

ProB random animations allow BTestBox to choose a number of random operation calls that we intend to use in our test cases. BTestBox executes the code translated following the given ProB animation history, calling each operation in the same order. In the end, BTestBox generates a report. If the results are equal, the tests pass, if they are not equal, the tests fail.

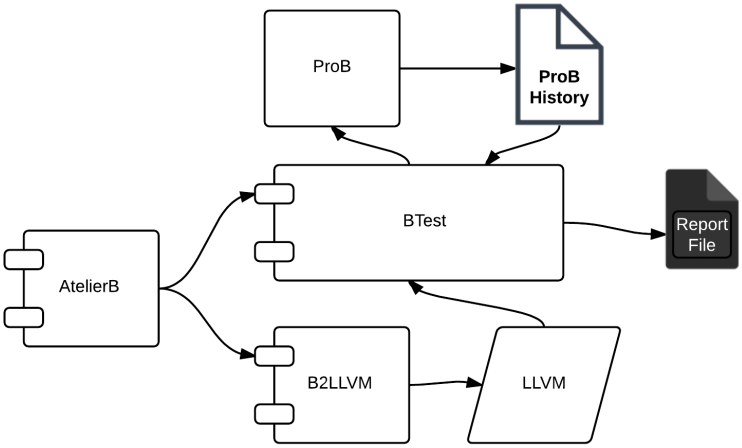


Figure 2. UML component diagram depicting the architecture of BTestBox

In this paper, BTestBox is used to validate the translation of B to LLVM Intermediate Representation (IR) done by the B2LLVM compiler. This case study is presented in section 4. B2LLVM is a compiler for the B implementation language that we have integrated to AtelierB[2]. It takes as input an XML file produced by AtelierB representing a B implementation and produces as output LLVM-IR format files. This process is presented in Figure 2.

To use BTestBox and verify the translations, we experiment with computers using Ubuntu 14.04 and OS X 10.11.3. To follow our testing procedures some essential tools beyond the ones provided by the OS are needed, such as LLVM-3.5, Clang-3.5 and *make*. LLVM refers to a multiplatform compiler infrastructure. Clang is a frontend compiler to C, C++, Objective C and Objective-C++ languages that uses LLVM as backend. Clang accomplishes fast compilations with low memory usage; besides, it shows diagnostics of errors in a detailed way. *make* is well-known tool dedicated to program builds.

3.2. How it works?

The interface of BTestBox is presented in Figure 3. In the *Maximum case tests* field the user choses how many operations will be called in ProB to animate the B implementation. In the field *Language of test script*, the user selects the target language. ProB has some options to change how the random animations are generated: the field *Heuristic test strategy* allows to set these options. Currently, BTestBox only implements tests using LLVM as a target language and does not support different coverage criteria. In the future, the tool shall support testing of C code and different criteria such as Modified Condition/Decisive Condition (MC/DC) and ACC (Active Clause Coverage). When the user finishes defining the test parameters, he just needs to press the *Generate* button and wait for the results.

4. Example/Case study

For the validation of BTestBox, we performed a case study which evaluated the B2LLVM compiler. In this case study, large, artificial, B models were produced and employed to

[2]This integration is not yet part of the standard distribution of AtelierB. Clearsy plans to deliver this functionality in version 4.5 of AtelierB.

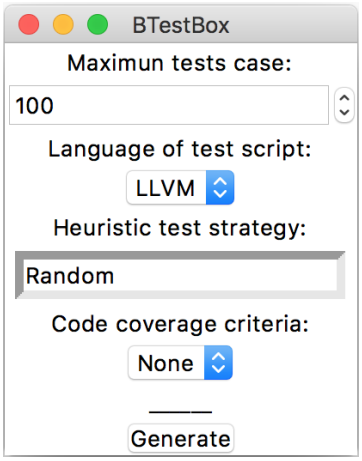


Figure 3. BTestBox Interface

```
IF c THEN  i END
```

```
WHILE c DO  i END
```

Figure 4. One generic instruction (While and If)

check the translation of B to LLVM. To generate these B models, B instructions are combined hierarchically and sequencially with different sizes, to form so-called instructions blocks. The number of instructions per blocks can be chosen, for this case study either 1 or 50 instructions per block. These instructions are combined using distinct depth levels. The tests created were based on [McKeeman 1998]. More specifically, this experiment uses simple test conditions over few variables, but with a large number of tests. These type of tests is also justified in the work of [Fischer et al. 2011].

Figure 4 presents a conditional and a repetition instruction. A simplified representation of these structures is found below, where c and i respectively symbolize the conditional expression and instruction. If i is substituted with a generic instruction in the if clause, this results in a composed instruction with a deeper nested instruction. The same occurs if the i in a while clause is substituted. Figure 5 illustrates this.

The predicate gets more complex each time a new substitution is applied in a generic instruction. The sentence is completed using basic instructions, each being well defined and manipulating a variable from the model, as seen in Figure 6. All conditional expressions are simple and vary the paths of the model execution, but will never result in an infinite loop. Every instruction has the conditional expression previously defined.

The B models generated have operations with many combinations of generics and basics instructions. The number of operations of the model is $NumOper = NumBInst*$

```
IF c THEN  IF ... END END
```

```
IF c THEN  WHILE ... END END
```

```
WHILE c DO  IF ... END END;
```

```
WHILE c DO  WHILE ... END END
```

Figure 5. Two generic nested instructions (While and If)

```
counter = (counter +1) mod 1024
```

Figure 6. Example of basic instruction

$NumGInst^{Depth}$, where $NumOper$ is the number of operations, $NumBInst$ is the number of basics instructions, $NumGInst$ is the number of generic instructions and $Depth$ is the nesting level. There are eight generic instructions and five basics instructions.

A model generator is used to create all the B machines and implementations. It produced B implementations with 10 million lines of code and 255 megabytes. Due to memory limitations it was not possible to create larger implementations. These implementations are much bigger than the largest examples found in literature, which are hardly larger than one megabyte. Some generated models were not even possible to test, due to limitations of the BXML compiler from AtelierB on our computers. Table 1 presents the metrics of generated models. Cases with an asterisk (*) were not possible to be evaluated due to these restrictions.

	Blocks	**1**				**50**			
		Btes Time	**B2LLVM Time**	**Number of Lines**	**File Size**	**Btest Time**	**B2LLVM Time**	**Number of Lines**	**File Size**
Depth	1	12,223 s	0,177 s	709	16 K	14,983 s	4,549 s	25209	480 K
	2	34,461 s	0,601 s	3509	64 K	454,136 s	23,178 s	126435	2,4 M
	3	343,163 s	5,853 s	26895	592 K	*	*	1096218	26 M
	4	3300,845 s	57,644 s	228456	5,4 M	*	*	10208798	255 M
	5	*	*	4475009	110 M	*	*	*	*

Table 1. B models metrics to evaluate the code generation.

The validation of the translation of the generated models was done comparing the execution of translated code (LLVM) with the ProB random animation. BTestBox generated tests in C that fulfilled this comparison. This approach provides a large set of tests and is totally automatic.

The validation process of the large models identified some limitations. The translator can not generate code to a VAR instructions nested into another due to an error of declaring the same variable two times in nested scopes. This problem was identified at compilation time, and did not generate a final executable code, so it does not offer any risk and was reported to be fixed. A limitation of BTestBox is related to the need of main memory and processing time. Another problem is that models not supported by BXML are also not supported by BTestBox. This can be a problem to apply BTestBox to code generators that do not rely on BXML, but this is not the case of B2LLVM.

For every program compiled without run-time error by B2LLVM, all test cases generated by BTestBox produced the expected results. For validation, 500 operation calls, or test cases, were used as default, for all B models, the motive being that our objective is to apply one metric to evaluate all models. Complementarily, 10 thousand test case were produced in 145 seconds for the smallest model. These tests are rapidly executed, in 90 milliseconds. We also manually inserted errors in the generated code and they were uncovered by BTestBox, giving us confidence that the whole procedure is functioning as expected.

5. Related Work

The automatic generation of test cases from formal models is a topic that has been targeted by several research groups. In [Marinescu et al. 2015], the authors presented an overview of the current state of the art for model-based testing tools that use requirement-based specification languages. In this survey, two tools for the B Method are presented: BZ-TT [Ambert et al. 2002] and ProTest [Satpathy et al. 2005]. BZ-TT is a tool-supported approach that generates test cases from B and Z models. It relies on constraint solving, and its goal is to test every operation of the system at every reachable boundary state. A boundary state is a system state where at least one of the state variables has a maximum or minimum boundary value. BZ-TT is a private tool and is not available to the public. ProTest is an automatic test environment for B specifications. It uses model-checking techniques to find test sequences that satisfy its test generation parameters. The user has to define the requirements to be satisfied by the test cases. These requirements are operations that must be covered and predicates that must hold true at the end of each test case. The tool only generates abstract test cases which have to be implemented before they can be executed; BTestBox is capable of generating executable test scripts, which is an advantage. Also, BTestBox make tests over a implementation and Protest over a model, our tool creates concrete tests directly in a programming language and is closer of the real code. Another recent contribution to this research field is the BETA tool [Matos 2016]. BETA relies on input space partitioning and logical coverage criteria to generate unit tests from abstract B machines. The tool automates all steps of the test generation process, from generation of test data to test data concretization and test script generation. Differently from BETA, our tool generates test cases directly from implementation modules which are a closer representation of the actual software. Another difference it that BETA is focused on unit testing, while BTestBox tests an entire module and its functions.

6. Conclusions and Future Work

BTestBox automatically generates tests to validate the output of code generators from B implementations. BTestBox can identify several types of problems such as operations in the translated code leading to a different state than what is foreseen in the B implementation (and it shows the expected and the actual state to the user), or more general problems like poorly defined implementations. BTestBox is now a fully functioning prototype, providing a novel and useful addition to the existing tools that support the B method. We plan to add several features and improve its usefulness and applicability: generation of code coverage metrics, support for Modified Condition/Decisive Condition coverage (the Federal Aviation Administration uses MC/DC, and some certification standards require this type of coverage); report of run-time metrics such as execution time and memory usage.

For this paper, LLVM was used as a target language for the translation, but adding other the target languages is possible with little effort, and we are also planning to do it, the goal being to extend BTestBox with as many languages as possible. Candidate programming languages are C and ADA, since AtelierB already has integrated code generators for these languages. We are also planning possible integrations with BETA [de Matos and Moreira 2012], another test generation tool for the B Method. Regarding other experiments, we are planning to perform experiments using mutation testing to evaluate the quality of the test cases generated by the tool.

References

Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2002). BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *Proc. of FATES 2002*, pages 105–120.

Bjørner, D. and Henson, M. C. (2007). *Logics of specification languages*. Springer Science & Business Media.

Chen, T. Y., Kuo, F.-C., Liu, H., and Wong, W. E. (2013). Code coverage of adaptive random testing. *Reliability, IEEE Transactions on*, 62(1):226–237.

de Matos, E. C. and Moreira, A. M. (2012). Beta: Ab based testing approach. In *Formal Methods: Foundations and Applications*, pages 51–66. Springer.

Dijkstra, E. W., Dijkstra, E. W., Dijkstra, E. W., and Dijkstra, E. W. (1976). *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs.

Fischer, B., Lämmel, R., and Zaytsev, V. (2011). Comparison of context-free grammars based on parsing generated test data. In *Software Language Engineering*, pages 324–343. Springer.

Leroy, X. (2008). The compcert verified compiler, software and commented proof.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Leuschel, M. and Butler, M. (2003). Prob: A model checker for b. In *FME 2003: Formal Methods*, pages 855–874. Springer.

Marinescu, R., Seceleanu, C., Le Guen, H., and Pettersson, P. (2015). A research overview of tool-supported model-based testing of requirements-based designs. Advances in Computers. Elsevier.

Matos, E. C. B. (2016). *BETA: a B based testing approach*. PhD thesis, Federal University of Rio Grande do Norte, Natal.

McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal*, 10(1):100–107.

Medeiros Jr., V. (2016). *Método B e a síntese verificada para código de montagem*. PhD thesis, Federal University of Rio Grande do Norte, Natal.

Medeiros Jr., V. et al. (2009). Aplicação do método b ao projeto formal de software embarcado.

Menzies, T. and Cukic, B. (2000). When to test less. *IEEE Software*, 17(5):107.

Rushby, J. (2008). Verified software: Theories, tools, experiments. *Automated Test Generation and Verified Software*, pages 161–172.

Satpathy, M., Leuschel, M., and Butler, M. (2005). ProTest: An Automatic Test Environment for B Specifications. *ENTCS*, 111:113–136.

Stuermer, I., Conrad, M., Doerr, H., and Pepper, P. (2007). Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM.

Sam: a Tool to Ease the Development of Intelligent Agents

João Faccin, Juei Weng, and Ingrid Nunes

[1]Universidade Federal do Rio Grande do Sul – Porto Alegre – Brazil

{jgfaccin,juei.weng,ingridnunes}@inf.ufrgs.br

Abstract. *Developing intelligent agents is a difficult task. The BDI architecture provides agents with flexible behaviour and our previous work extended this architecture by using learning to improve plan selection. In this paper, we present Sam, a tool that supports BDI agent design and implementation, using a model-driven approach. Sam provides an environment where agents can be modelled according to our proposed meta-model, in which developers focus on domain-specific concepts. Our tool, based on a design model, also generates agent code focusing on the BDI4JADE platform. Our approach hides from developers sophisticated techniques, so that mainstream software developers are able to leverage such techniques without having to learn their technical details. Video:* https://youtu.be/hkftWZx82EM

1. Introduction

Software has shifted from standalone applications to distributed and highly interactive systems, evidenced by cloud computing and smart grids. Such complex systems consist of a composition of autonomous software components, with proactive and reactive behaviour, and social ability. In the context of multi-agent systems [Wooldridge 1999], such components are referred to as *agents*, and much work has been done to address the problems that emerge in this scenario. For example, agents must be coordinated, learn in which other agents they can trust, and adapt themselves according to its context. Agent architectures have been proposed to support agent development. One of the most widely used architectures is inspired by human practical reasoning, namely the BDI (*belief-desire-intention*) architecture [Rao and Georgeff 1995]. In this architecture, goals are explicitly represented, and agents have a reasoning cycle that includes the selection of appropriate actions (plans) to achieve goals. Consequently, agent behaviour is flexible in that it can select plans that are suitable to the current context or execute other plans in case of failure.

The BDI architecture is abstract, and there are gaps that must be fulfilled to provide agents with intelligent behaviour. Our previous work [Faccin and Nunes 2015, Faccin 2016] proposed a model-driven approach, which fills one of these gaps with the provision of a learning-based plan selection technique. The key goal was to provide means for developing BDI agents able to learn in which context plans perform best, thus making a wiser plan selection. In this work, we present the Sam tool, which supports the use of our technique by providing an environment to model BDI agents, focusing on domain aspects, and generate agent code. Consequently, we free developers from learning complex learning models or details of the BDI reasoning cycle.

2. Background on BDI Agents and Learning-based Plan Selection

BDI agents are composed of three key components. The first are *beliefs*, which represent its current state. They are referred to as beliefs, rather than attributes, because they may

not be accurate, e.g. due to environment perceptions with noise. The second are desires, or *goals*, which represent something the agent wants to achieve. The third are intentions, which are goals that an agent is committed to achieve and already has a *plan* to execute to achieve it. Its reasoning cycle, which is provided by BDI platforms, is the following: (1) beliefs are revised according to perceived events; (2) goals are updated according to current beliefs (goals may have been achieved or no longer desired, or new goals are created); (3) a subset of goals are selected to be achieved (goals may conflict with each other, so an agent may select a subset of them); and (4) a plan from a set of candidates is selected to be executed to achieve each intention. Note that if the selected plan fails, the agent still has the goal, so other options are explored to achieve it. Step 4 is referred to as *plan selection*, and its default implementation randomly selects a plan from those whose context condition matches the current context.

Our plan selection approach is composed of a *meta-model* and a *technique*. The meta-model specifies concepts to model agent preferences, softgoals and plan metadata. Making an analogy to software development, *softgoals* can be seen as non-functional requirements. A simple example is an agent that has the goal of sorting an array, and softgoals indicate other properties to be satisfied, such as time taken and used memory. Plan metadata consist of information to understand which *factors* influence plan *outcomes*, e.g. the number of added items since the last time the array was sorted influences the time taken by a sorting algorithm (each is a plan). Finally, preferences allow specification of softgoal importance to an agent. The technique, based on this information, builds a prediction model based on observations of plan executions. Using this model, it predicts plan outcomes based on the current context. Finally, it calculates plan utility considering agent preferences and estimated outcomes. The plan with the highest utility is selected.

3. The *Sam* Tool

Sam is a development environment that includes a graphical editor and code generator developed focusing on the design and implementation activities of the agent development process. Implemented as an Eclipse[1] plug-in, it allows users to graphically instantiate agent models, also providing features for automatic validation and code generation of these models. In this section, we detail Sam, presenting its main features, the different elements that comprise its user interface and, finally, its architecture.

3.1. Features and User Interface

Our tool provides two key functionalities. The first provides assistance for users to design and model software agents. Therefore, Sam includes a modelling editor to design agents, which is presented in Figure 1. The main element of the Sam user interface is the editor view, which is responsible for displaying the graphical representation of a modelled agent. The information related to a modelled element is stored in diagram and model files, related to graphical and domain-specific information, respectively. Users can navigate these files through the package explorer, located on the left-hand side of the editor view. Below the package explorer, an additional view presents a miniature of the model being presented by the editor view, thus assisting users in visualising large model representations. On the right-hand side of the editor view, there is a creation palette, which contains the elements that can be instantiated in a diagram. These elements, when instantiated, can have their properties changed through a property view, positioned below the

[1] `http://www.eclipse.org/`

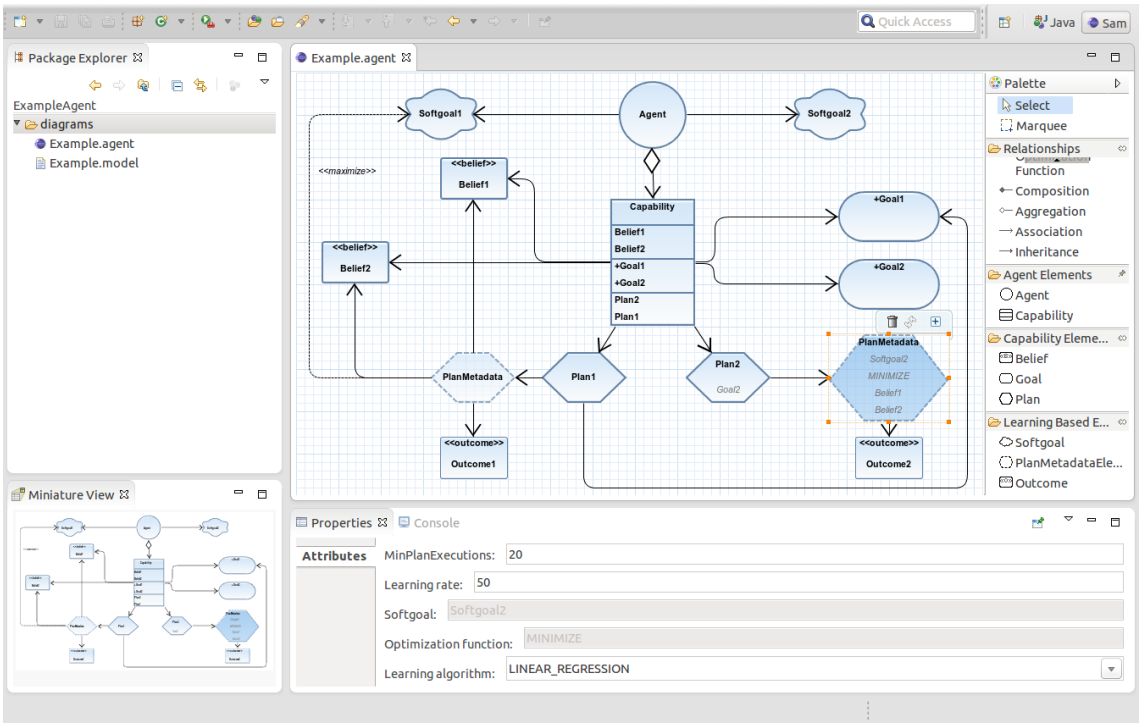


Figure 1. Sam Overview: the Modelling Editor.

editor view. This property view can also be replaced by a console view, which is used to provide specific feedback to users.

The second key functionality supports agent implementation, by generating code according to an instantiated agent model. This code generation feature is automated, and it requires users only to request to generate code. Note that generated code is not limited to simple code skeletons, but includes code to reason about plans. Next, we detail these modelling and code generation features.

3.1.1. Agent Modelling

The modelling feature that Sam provides allows the graphical instantiation of agents following our introduced meta-model. This meta-model defines which elements comprise an agent, and how these elements are related to each other. Our current implementation of the meta-model also includes other extensions of the BDI architecture [Nunes 2014].

In Sam, instantiating a model involves a few steps. Initially, users must create a new diagram file using a specific creation wizard that we provide. Thus, they are able to model a new agent by dragging elements from the creation palette and dropping them into the diagram area presented by the editor view. Graphical representations of the elements composing our meta-model are described as follows. An agent is represented by a circle containing the agent name. Agents, in our meta-model, are an aggregation of capabilities, which are components that modularise a set of beliefs, goals and plans. Capabilities are represented by rectangles with four compartments, which contain its name, and the name of its beliefs, goals, and plans, respectively. A belief is depicted as a rectangle containing a stereotype ≪`belief`≫ and the belief name. Goals are depicted as rounded rectan-

gles while softgoals are represented by cloud-like shapes, both containing the respective element name. Plans and plan metadata elements are represented by hexagons with solid and dashed lines, respectively, also containing the element name. Such plan metadata elements are related to outcomes whose representation is similar to that of beliefs, but with the stereotype ≪outcome≫.

Relationships between elements also have their particular representations. Association, aggregation, composition and inheritance relationships are illustrated in the same manner they are in UML class diagrams—solid directional lines with particular arrowheads connecting two elements. Optimisation functions, in turn, are presented as dashed directional lines labelled with the name of the given optimisation function. These functions are used to convert plan outcomes into preference values. A preview of the graphical representation of each available element or relationship is presented in the creation palette. Moreover, it is important to highlight that, to maintain consistency with existing agent methodologies, some representations were imported from those used by Tropos [Bresciani et al. 2004].

Sam provides several usability features, aiming to allow users to create understandable models while improving their experience using our tool. When modelling an agent that aggregates several capabilities, users can create a separate diagram file for each of them. Such capability diagrams can be associated with a particular agent diagram file. Given that our tool is able to identify every component belonging to the same model file, every change performed in a capability diagram reflects on its associated agent diagram. Therefore, users are able to maintain a simplified agent diagram containing only representations of an agent, its capabilities, and their relationships. The representation of capabilities and their elements are, in turn, maintained in their respective capability diagrams. Additionally, users can navigate these related agent and capability diagrams using the so-called *drill-down* feature. This feature simply opens the diagram to which a capability is related when an *Open associated diagram* option is selected in the capability's context menu. Moreover, it is bidirectional, i.e. it is possible to go from an agent to a capability diagram, and from a capability to its agent diagram. Figure 2 illustrates the Sam drill-down feature.

Another feature that contributes to simplifying the visualisation of larger model representations is the *collapse* feature. It allows users to hide/show graphical representations of particular relationships whose source is a plan or plan metadata element. When hidden, the target name of a given relationship is presented inside the source element. When shown, the target name disappears from the source element representation and the relationship shape becomes visible again. Such feature can be activated using the collapse context button, which is shown when the mouse is positioned on a plan or plan metadata element. An example of the collapse feature activated can be observed in Figure 1. Finally, we also provide the validation of relationships between elements, e.g. if a plan is already associated with a goal, the user becomes unable to insert a new relationship with a different goal unless the previous one is removed.

Agent models are stored in two kinds of files: (i) the diagram file, containing details of the graphical representation of the agent model, and (ii) the model file, which stores the model itself. While the former is used to correctly display the model, the latter becomes the input to the code generation process, detailed next.

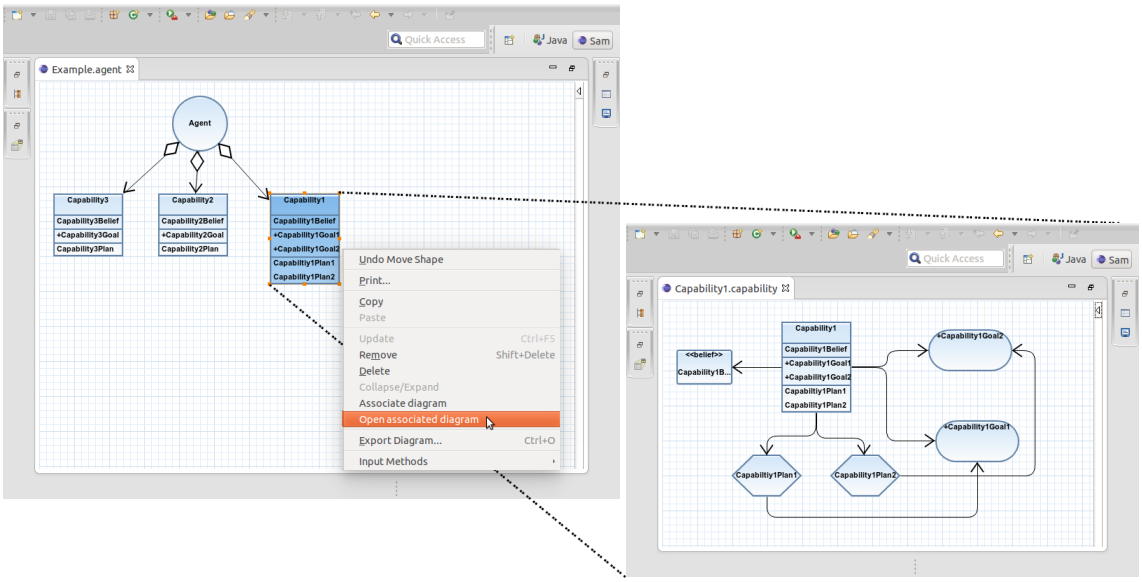


Figure 2. Navigating through Diagrams with the *Open associated diagram* option.

3.1.2. Code Generation

The code generation feature provided by Sam is responsible for creating code from agent models. By taking a model file as input, our tool is able to automatically generate BDI4JADE [Nunes et al. 2011] code corresponding to the agent represented in such model, as well as its entire structure of packages and classes, and required libraries. BDI4JADE is a BDI platform implemented in Java, which was selected because it is concerned with the industrial adoption of the agent technology. Before starting the code generation process, Sam checks the model looking for inconsistencies. Any inconsistency found is reported to users through the console view. There are two classes of model inconsistencies: (i) *regular*, which does not affect code generation; and (ii) *severe*, which immediately interrupts the code generation process. Regular inconsistencies are related to missing information that can be added directly to the code later, while severe inconsistencies refer to missing relationships or elements that are required to be instantiated on the given model, otherwise the generated code would be semantically wrong.

Users can trigger the code generation feature by selecting the *Generate Code* option in the context menu of a model file. Thus, Sam requires users to select one of two available agent profiles: *standard agent* or *learning-based agent*. Each profile corresponds to particular model validation and code generation approaches. The *standard agent* profile refers to typical BDI agents and those implementing capabilities relationships [Nunes 2014], while the *learning-based agent* profile indicates to Sam that it must check and generate code of agents that implement our learning-based approach. Figure 3 illustrates Sam's code generation feature. It is important to highlight that Sam only generates the code it can infer from a model file, e.g. correctly initialising elements and combining them with code that implements our learning-based technique. Therefore, developers should complete some of the agent components, mainly plan bodies, with the domain-specific code that cannot be generated automatically.

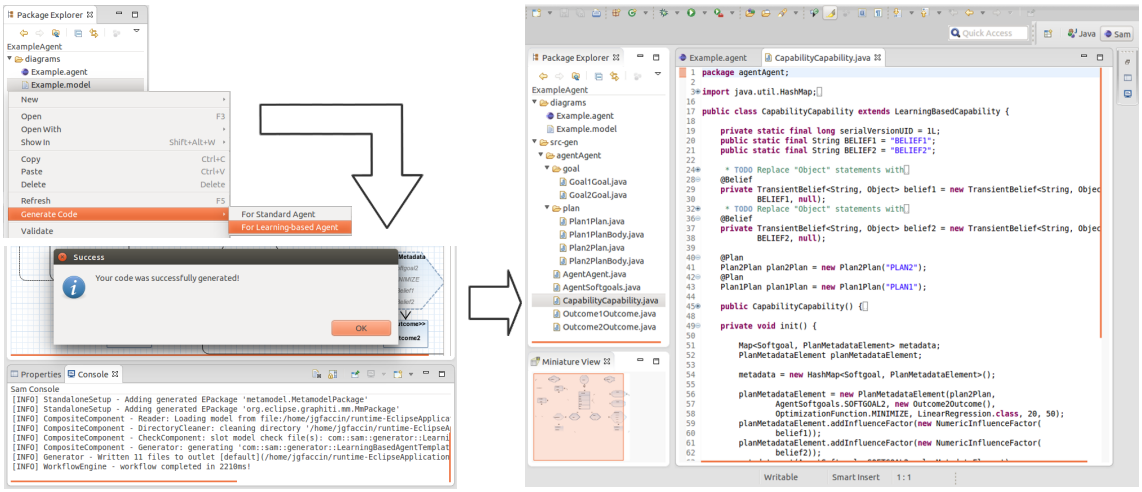


Figure 3. Code Generation Feature.

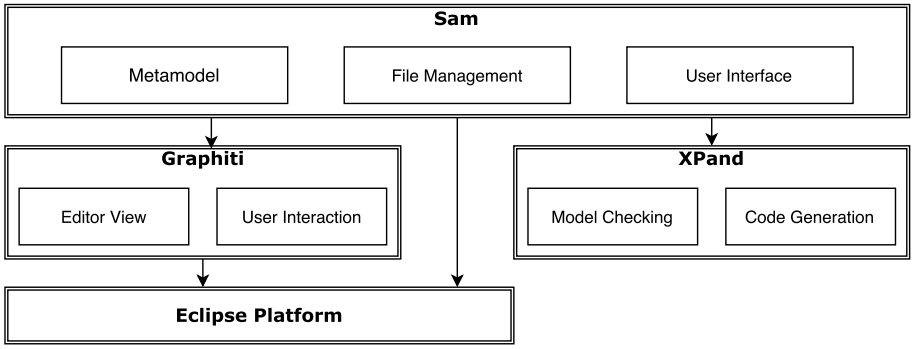


Figure 4. Architecture Model.

3.2. Sam Architecture

Our tool is developed as an Eclipse plug-in built upon the Graphiti[2] framework and the XPand[3] template language. The integration between these different technologies and our tool is presented in Figure 4. Graphiti is particularly suitable for developing graphical editors and viewers. Therefore, it is responsible for creating and displaying the editor view and for handling the user interaction with it. XPand, in turn, is a template language whose functionalities are used specifically on Sam's code generation process. It is responsible for the model validation and code generation. Finally, Sam aggregates the classes related to the meta-model used as the basis for agent modelling and works together with the Eclipse platform to manage diagram and model files and the user interface that does not correspond to the editor view.

4. Evaluation

Previous work that is underlying the Sam tool was evaluated with simulations and empirical studies [Faccin and Nunes 2015, Faccin 2016]. We also evaluated Sam empirically considering three aspects: (i) tool usability, (ii) user satisfaction, and (iii) ease of

[2]http://www.eclipse.org/graphiti/
[3]http://wiki.eclipse.org/Xpand

use. A group of 11 voluntaries was requested to perform two different activities involving the use of our tool. After performing these activities, they were asked to answer a questionnaire—adapted from the USE (Usefulness, Satisfaction, and Ease of Use) questionnaire [Davis 1989, Lund 2001]—composed of 26 seven-point Likert rating scales and two open-ended questions. For each question, voluntaries assigned a score ranging from 1 (strongly disagree) to 7 (strongly agree) to a given statement regarding one of the three aspects evaluated. We also requested voluntaries to provide a list of the three most negative and three most positive aspects they experienced when using our tool. The complete set of questions, as well as obtained answers and additional information on the study participants, are available elsewhere [Faccin 2016].

Results indicate that voluntaries found Sam to be useful ($M = 6.18$, $SD = 1.08$), would recommend it to a friend ($M = 5.45$, $SD = 1.63$) and believe it is easy to remember how to use it ($M = 5.64$, $SD = 1.12$). Although there is no agreement that the tool does everything they would expect it to do ($M = 4.45$, $SD = 1.51$), the majority of voluntaries stated that Sam is somewhat pleasant to use ($M = 4.82$, $SD = 1.83$). Considering answers to the most positive and most negative aspects experienced while using our tool, some voluntaries mentioned that it was easy to get started; lots of useful code is automatically generated, and the tool minimises code handling. However, some voluntaries pointed out some problems. They mentioned that model inconsistencies were not shown or easy to detect, and that a model diagram may become confusing for larger projects.

As this evaluation was performed with a previous version of our tool, we used its results to improve Sam. We were particularly concerned with the issues involving model inconsistency detection and the scalability of the notation used in models. The former was addressed by adding model validation before code generation, as mentioned earlier. The drill-down and collapse features were introduced to deal with the scalability issue.

5. Related Work

There are several approaches that propose the use of graphical editors to model and generate code for agents and multi-agent systems. The DSML4MAS Development Environment (DDE) [Warwas and Hahn 2009] is a tool that supports modelling and code generation of agents developed using a domain-specific modelling language that allows developers to graphically model multi-agent systems with different abstraction levels. Gascueña et al. [Gascueña et al. 2012] present an approach addressing agents based on the Prometheus methodology [Padgham and Winikoff 2004]. For this purpose, they developed the Prometheus Model Editor, allowing developers to graphically model and generate code for Prometheus agents. Pavón et al. [Pavón et al. 2006] present a reformulation of a particular agent development methodology, where they use existing tools provided by such methodology to allow agent modelling and code generation, using different model-to-code transformations to address specific agent platforms.

These initiatives are limited to modelling traditional agent concepts, which do not include techniques that provide agents with intelligent behaviour. Consequently, generated code consists of code skeletons that are similar to code generated from UML class diagrams, and much is left to developers. Our approach, instead, hides sophisticated techniques from developers, so that mainstream software developers are able to leverage such techniques without having to learn their technical details. Sam also provides additional features, such as those to deal with scalability (drill-down and collapse features).

6. Conclusion

Developing software agents is not a trivial task. Frequently, the need for understanding complex concepts and techniques underlying such technology contributes to keeping away users that could potentially benefit from its use. In this paper, we presented Sam, a tool to support the design and implementation of agents with learning capabilities. It provides modelling and code generation features that, when used together, are able to deliver an almost complete BDI4JADE agent code. Sam was not only developed to ease the development process of intelligent agents, but also to promote the adoption of the agent technology in the industry. Our tool will be publicly available with the next stable version of the BDI4JADE platform, licensed under LGPL.

Acknowledgements

This work receives financial support of CNPq, project ref. 442582/2014-5: "*Desenvolvendo Agentes BDI Efetivamente com uma Abordagem Dirigida a Modelos*." Ingrid Nunes thanks for research grants CNPq ref. 303232/2015-3, CAPES ref. 7619-15-4, and Alexander von Humboldt, ref. BRA 1184533 HFSTCAPES-P.

References

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340.

Faccin, J. (2016). Preference and context-based bdi plan selection using machine learning: from models to code generation. Master's thesis, UFRGS, Porto Alegre. pages 61, 72.

Faccin, J. and Nunes, I. (2015). Bdi-agent plan selection based on prediction of plan outcomes. In *WI-IAT*, volume 2, pages 166–173.

Gascueña, J. M., Navarro, E., and Fernández-Caballero, A. (2012). Model-driven engineering techniques for the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 25(1):159–173.

Lund, A. M. (2001). Measuring usability with the use questionnaire. *STC Usability SIG Newsletter: Usability Interface*.

Nunes, I. (2014). *Engineering Multi-Agent Systems: Second International Workshop*, chapter Improving the Design and Modularity of BDI Agents with Capability Relationships, pages 58–80.

Nunes, I., Lucena, C. J. P. D., and Luck, M. (2011). Bdi4jade: a bdi layer on top of jade. In *International Workshop on Programming Multi-Agent Systems*, pages 88–103.

Padgham, L. and Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. New York, NY, USA.

Pavón, J., Gómez-Sanz, J., and Fuentes, R. (2006). *Model Driven Architecture – Foundations and Applications: Second European Conference*, chapter Model Driven Development of Multi-Agent Systems, pages 284–298. Berlin, Heidelberg.

Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: From theory to practice. In *International Conference of Multi-Agent Systems*, pages 312–319.

Warwas, S. and Hahn, C. (2009). The dsml4mas development environment. In *AAMAS*, volume 2, pages 1379–1380, Richland, SC.

Wooldridge, M. (1999). Intelligent agents. In *Multiagent Systems*, pages 27–77.

AMT: An Android Mirror Tool for Instant Feedback Across Platform

Eduardo Noronha de Andrade Freitas[1], **Celso G. Camilo-Junior**[2], **Kenyo Abadio Crosara Faria**[1], **Auri Marcelo Rizzo Vincenzi**[3]

[1]Departamento de Informática – Instituto Federal de Goiás (IFG)
Goiânia – GO – Brazil

[2]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

[3]Departamento de Computação
Universidade Federal de São Carlos (UFSCAR) – São Carlos, SP – Brazil

{efreitas,kenyofaria}@ifg.edu.br, celso@inf.ufg.br, auri@dc.ufscar.br

Abstract. *The worldwide smartphone market keeps growing each year and, according to International Data Corporation (IDC) [IDC 2015], Android continues to dominate the global smartphone market nearly 82.8% of the market share in 2015. However, there are more than 24 thousands of different devices available in different screen sizes and densities and other configurations. While fragmentation may be considered a positive point, the development of applications for Android has been challenged, especially regarding to software testing activities. Android fragmentation is a problem that has long concerned software developers, and thus, we investigated what could be a possible way to easily check an Android app compatibility across devices to assist those developers to minimize the market vulnerability. With this inspiration, we present Android Mirror Tool (AMT), as an alternative for checking the compatibility of user's interactions across platform. AMT allows the users to observe interactions made on a single device to be replayed in a set of others devices, revealing instantly incompatibilities. AMT is also able in generating scripts for future reproduction in different API's such as UIAutomator and Espresso API. The results of our experiments using nine-top Android apps showing the potential and benefits of using AMT.*

URL video: https://youtu.be/LEGEZ8aKZ6M

1. Introduction

Despite the fact Android continues dominating the global smartphone market [IDC 2015], the android fragmentation is a problem that has affected and worried many developers around the world. Fragmentation has been considered both a strength and weakness of the Android ecosystem. While android provides many alternatives for the user, there are a great number of design and testing minefield with hundreds of device screen sizes, hardware features, OS versions, input gestures and just about every other testing scenario you could imagine. Thereby, the developers/testers need to make sure that their apps will run properly on those devices. However, such verification task is quite complex because

testing all different configurations is almost impossible in a practical way besides being very expensive.

The task of checking the compatibility of an android app across platform has been performed based on three different approaches: 1) manually which is expensive and fault prone; 2) writing scripts (e.g., monkeyrunner syntax) or UI test cases (e.g., UIAutomator API, or Espresso API [Espresso 2016]) which involves high costs (creation and execution) depending to the criteria chosen; and 3) capturing events on a source device for a postponed execution in different devices (RERAN [Gomez et al. 2013], Testdroid recorder [Kaasila et al. 2012], GUITAR [Hu and Neamtiu 2011], Monkey Talk [Kipar et al. 2014], and ACRT [Liu et al. 2014]).

We list some of the main advantages of using AMT against those tools: (1) captured scripts on those tools can be replayed only on the same device where they were captured because they are based on x,y coordinates while AMT allows reproduction in different devices; (2) at the end of the capture process, AMT is able in generating input test case written in Espresso API; (3) AMT runs in real time environment allowing to see instantly incompatibility across devices; (4) No installation or instrumentation process is required on the devices when we make use of AMT.

AMT does not need the source code, and it does not perform intrusive instrumentation or similar process on the devices.

AMT provides a view of how much an app is vulnerable in terms of its compatibility on the Android market by simply executing a set of user's interactions on a device and seeing instantly how these actions work across devices. To the best of our knowledge, AMT is the first tool using mirror approach for Capture/Replay techniques in real time over multiple devices. Also, the first tool to generate automated inputs of UI testing for Espresso [Espresso 2016]. In Section 2 we present the concept of market vulnerability used by AMT, in Section 3 we present the technique behind AMT. Our empirical evaluation is presented in Section 4, and finally we present our conclusions in Section 5.

2. Android Market Vulnerability

Android fragmentation is a problem that has long concerned software developers. Android owns a proliferation of brands corresponding to approximately 24,000 different devices, four generalized screen sizes (small, normal, large, and extra-large), six generalized densities (ldpi, mdpi, hdpi, xhdpi, xxhdpi, and xxxhdpi), 23 operating system versions. To illustrate, taking into consideration these features and nine different sizes of memory, we yield the possibility of 4,968 different device configurations.

The market vulnerability metric was created to express the vulnerability of a user interaction across devices according to market distribution [Android Market 2016]. In sum, the greater the market penetration of a given device configuration, the greater the probability that failures on apps running on such device will affect many users. Thus, if a user interaction has a failed execution, the devices in which the execution failed are identified, and the market vulnerability is defined in terms of the market share of that device.

Table 1 presents an example in which one component failed in three devices (D3, D5, and D10). In this case, the market vulnerability (mv) is 66.45%, and it was computed

by $am + (s/2)$, where am is the API market, and s is the screen / density market share.

Table 1. Example of method market vulnerability.

Device	API	Screen Size	Density	API Market	Screen / Density Market
D3	21	Xlarge	xhdpi	16.2%	0.7%
D5	19	Normal	xhdpi	32.5%	24.9%
D10	18	Normal	mdpi	2.9%	4.1%
			Total	51.6%	29.7%

3. Android Mirror Approach

In this section, we first present the technique behind AMT, and after that we provide some main characteristic details of our implementation.

3.1. Technique Overview

In the Algorithm 1 is presented an overview of our technique. Basically, the process takes as input user's events in a source device, captures and processes these events, generates a high level language based on these user's events, and sends this information to all devices/emulators on the platform to be replayed. As a result of this process, a set of *monkeyrunner* commands for each emulator and a set of input tests for Espresso API are presented as output. Our technique can be described by the following three main phases: capture, code generation, and replay.

The capture phase starts by using an android application in a source device. All user's interactions with on source device are monitored using *getevent* tool (available on Android SDK) that provides information about input devices and a live dump of kernel input events, as represented by the variable ue at the line 2 in the algorithm. A parser in the ue is processed looking for some details of the events such as (x,y) coordinates, kind of event, and timestamp information (line 3).

Furthermore, a dump of the current UI hierarchy is analysed (line 4) with mirror event (ME) to identify what was/were the UI object(s) manipulated in the user event. We matched (line 5) the UI object using (x,y) coordinates. The algorithm tries to figure out, on the UI tree, an element that matches with some properties: id, class name, package name, text, and content description. If the correspondent node is not found in the UI tree, the algorithm finds the UI element using a *xpath* obtained in the capture phase. As a result of this step, a set of information called by high level event (HLE) can be used for different purposes (e.g. replay, code generation). In the line 6 HLE is published to be consumed for all replay devices/emulators, and in line 9 the UI input test in Espresso is generated (more details in the subsection 3.2.3).

Each emulator on the platform is represented by a replay agent that consumes messages (HLE) sent by the capture agent. We implemented AMT on top of a publish/subscriber service. For each event received by a replay agent, a dump of the UI Hierarchy is done (line 14), and a reverse mapping in the capture phase is realized by taking HLE and trying to figure out what are the correspondent (x, y) coordinates of the UI Object(s) in the replay emulator (line 15). This process is necessary once that monkeyrunner is oriented by (x, y) coordinates, and not by UI objects. All monkeyrunner commands are persisted in a file ($SMRC$) to be run hereafter (line 16).

Algorithm 1: Android Mirror Tool: Overall Algorithm

/* **Capture** */
Input : UE: Set of user events from android device
Output: HLE: High Level of Events defined by Android Mirror Tool

begin
  foreach $ue \in UE$ **do**
    $ME \leftarrow processUserEvents(ue)$
    $DUMP \leftarrow getDUMP()$
    $HLE \leftarrow matchDOM(ME, DUMP)$
    $Publish(HLE)$
  return HLE

/* **Input Test Generation for Espresso API** */
Input : HLE: High Level of Events
Output: ITE: Set of Input Tests for Espresso API

begin
  $ITE \leftarrow generateInputTestEspresso(ITE)$
  return ITE

/* **Replay across Different Emulators** */
Input : HLE, E: Set of emulators = $\{\mathcal{E}_1, \mathcal{E}_2, .., \mathcal{E}_n\}$
Output: $SMRC$: Set of monkeyrunner commands

begin
  $SMRC \leftarrow \{\ \}$
  foreach $\mathcal{E} \in E$ **do**
    $DOM \leftarrow get_DOM(\mathcal{E})$
    $MRC \leftarrow prepareMonkeyCmd(HLE, DOM)$
    $SMRC.add(MRC)$
  return $SMRC$

3.2. Implementation

AMT was written in Java, and can run on a variety of desktop operating systems, including Windows, Mac OS, and Linux. AMT was built to work with real android devices and also emulators such as Android Virtual Devices (AVD) and Genymotion [Genymotion 2016] emulators. Basically, AMT needs a source device, and a set of emulators to replay the user's interactions. The source device is used by the user to interact with an Android app. As a Java desktop application, AMT runs on a machine with JVM, and it requires that the source device and also the replay devices are connected with that machine. Once they are connected, the user can start to use an android application in the source device, and see the mirror behavior across platform. In terms of design, AMT has four main components: event converter, match engine, Espresso code generator, monkeyrunner command generator, as shown in Figure 1.

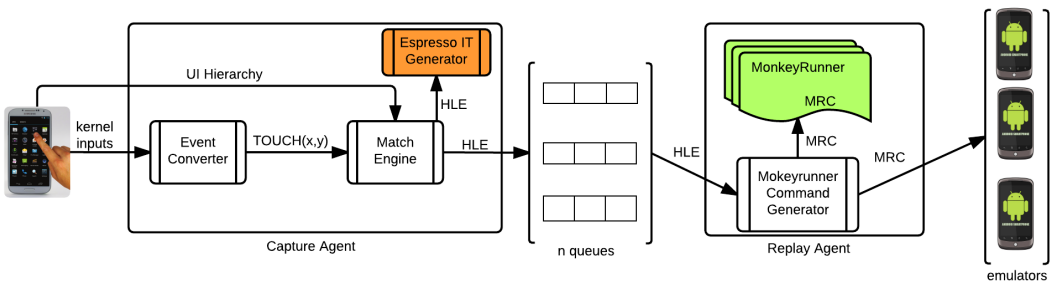


Figure 1. High Level design of Android Mirror Tool (AMT).

3.2.1. Event Converter

The first component is the event converter. AMT uses $getevent$ tool to capture all kernel input events, and interprets them. After, AMT generates a command containing an user's action and (x, y) coordinates (e.g., touch(420, 670)). The Android Debug Bridge (ADB) is a command line tool used to communicate AMT with our devices/emulators.

3.2.2. Match Engine

The Match Engine takes $uiautomator$ through its dump option to generate a xml file from the current UI hierarchy. Once AMT has both the xml file and the event information from the previous phase, the matching of the UI element on the xml file is done according to explaination in the section 3.1. As a result of this process, an object called HLE is generated. HLE is a defined format to share information between publish and subscribers in AMT. In a HLE there are information about UI properties, and kernel inputs.

3.2.3. Espresso Input Test Generator

Alternatively, AMT provides a way to generate input tests automatically in Espresso API. In this moment, the tool is prepared to generate automated input tests for two different API's: UIAutomator, and Espresso [Espresso 2016].

Espresso API was developed by Google and presents a simple API to the test author that focuses on making it easy, reliable, and durable. It requires to specify which UI element to interact with and then the type of interaction or assertion that should be performed on the element. Espresso transfers safely the state between the test thread and the UI thread, and delay evaluating the interactions until the UI has reached a point where it will not change without further user input [Knych and Baliga 2014].

The component responsible to generate Espresso input tests is Espresso Input Test Generator. Basically, it takes a HLE and writes a class according with Espresso API. In this class, we have a list of Espresso sentences corresponding to the user events done on the source device. All HLE are mapped in Espresso sentences. To generate these input test for Espresso, AMT was assisted by Java Writer [JavaWriter 2016]. Java Writer is an utility class which aids in generating Java source files.

3.2.4. Publish/Subscriber Model

Our tool was built based in a publish/subscriber model, where subscribers (represented by replay emulators) register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers (represented by a source device). AMT utilized Rabbitmq library [Rabbitmq 2016] to implement this distributed behaviour. Basically, we have two agents: publish agent (called capture agent), and subscriber agent (called replay agent). The capture agent is responsible for coordinating the three other components: Event Converter, Match Engine, and Espresso Code Generator. Also, it publishes on the queue of the model, the HLE to be consumed by

replay agent. On the other hand, the replay agent is a thread safe implementation that is responsible for consuming all published HLE. Each android device/emulator (except source device) registered on AMT is represented by a replay agent. Replay agent is also responsible for coordinating the Monkeyrunner Command Generator component.

3.2.5. Monkeyrunner command generator

The Monkeyrunner command generator takes a HLE and tries to figure out its relative (x,y) coordinates, applying a reverse method from Match Engine. After that, once having a (x, y) coordinates, AMT sends a Monkey Runner Command (MRC) to a file, and uses ChimpChat to execute these events on the emulators across the platform. ChimpChat is a host-side library that provides an API for communication with an instance of monkeyrunner on a device. The monkeyrunner tool provides an API for writing programs that control an Android device or an emulator from outside of Android code.

A great advantage of AMT is possibility of using emulators instead of real devices. Besides saving money, and avoiding the management of physical devices, the use of emulators can reveal a significant percentage of errors across UI testing, mainly exploring the availability of the cloud resources. In addition to permit the usage of real devices, AMT supports some Android Virtual Devices (AVD), such as Genymotion emulators.

4. Empirical Evaluation

In order to assess the efficiency of our approach we defined a research question to target our empirical evaluation. Can AMT be used to show the compatibility of android apps considering: android version, screen size, and density? We firstly describe the evaluation subjects, and then, we conclude by discussing the evaluation results.

4.1. Experimental Subjects

We considered 9 open source Android apps in our empirical evaluation. The selection of these apps was done based on the number of installations for an app from the Google Play store [Google 2016] on end-user Android phones. We prioritized apps with higher number of installations. We selected apps from different categories (as listed by the Google Play store) to have a diverse corpus of subjects. Table 2 lists the subjects used in our evaluation. For each subject, the table shows its subject name, name, category, the range of its installations (*Installations*), and the number of user's interactions.

4.2. Results

To avoid study bias in defining application use, 17 graduate students and seven developers/testers from the industry were requested to define user's interactions over our 9 subjects yielding 220 valid user's interactions. When they completed this task, we manually executed all of them in the master device to see the behavior in different devices cross the platform. As a result, AMT was able to generate a reproducible script based on the user's interactions, for the most user's interactions excepted in 13 where the participants used long click actions. So, AMT was efficient in 94.09% of user's interactions generated.

We also collected the results from the execution cross devices to check the level of compatibility of each app in different android devices. We executed the user's interactions

cross five emulators: LG G FLEX (D1), HTC ONE (D2), SONY XPERIA Z3 (D3), SAMSUNG GALAXY S5 (D4), and LG G3 (D5). All of them use API 19 with 39.1% of the market, according to the Table 3 and data available on [Android Market 2016].

Table 2. Description of experimental subjects.

Subject Name	APP Name	Category	Min Installs	Max Installs	Number of User's interactions
A1	Daily Money	Finance	500,000	1,000,000	28
A2	Alarm Klock	Tools	500,000	1,000,000	30
A3	Simple C25K	Health & Fitness	50,000	10,000	27
A4	EP Mobile	Medical	50,000	100,000	30
A5	BeeCount	Productivity	10,000	50,000	30
A6	Bodhi Timer	Lifestyle	10,000	50,000	30
A7	andFHEM	Personalization	10,000	50,000	15
A8	Xmp Mod	Music & Audio	10,000	50,000	15
A9	World Clock	Travel & Local	50,000	100,000	15
Total	–	–	1,190,000	2,410,000	220

Table 3. Real Devices used to measure compatibility rate.

Setup#	Screen	Density	Real Device	Market		
				Screen	Min	Max
D1	Large	hdpi	LG G Flex	0,6%	39,1%	39,7%
D2	Normal	hdpi	HTC One	38,7%	39,1%	77,8%
D3	Normal	xhdpi	Sony Xperia	18,9%	39,1%	58,0%
D4	Normal	xxhdpi	Galaxy S5	15,8%	39,1%	54,9%
D5	Large	xxhdpi	LG G3	0,0%	39,1%	39,1%

Each user interaction was performed 10 times cross-devices to compute the compatibility rate. Table 4 shows the compatibility rate for each app. In that table we have the percentage of errors found in all user interactions performed. Thus, the higher value at Table 4 means a lower compatibility. For instance, taking 30 natural language test cases from A4 executed on D2, 4 had failures. Based on these information, we were able to measure the app vulnerability for those devices (D1-D5) based on the provided user's interactions. Although we can clearly see some apps with a low rate of compatibility as the A7, the compatibility rate computed for A9 for example does not mean that this app is stable cross devices. Naturally, the requirement level of a user interaction can influence the compatibility rate, requiring more resilience from apps when the user interaction is more sophisticated, and permitting high compatibility if the user interaction is simpler.

Table 4. Compatibility risk cross-device.

	D1	D2	D3	D4	D5
A1	13%	7%	13%	20%	7%
A2	7%	7%	7%	13%	7%
A3	0%	0%	0%	14%	0%
A4	14%	29%	21%	29%	14%
A5	21%	14%	7%	14%	7%
A6	0%	0%	8%	15%	8%
A7	50%	36%	29%	57%	21%
A8	0%	0%	0%	7%	0%
A9	0%	0%	0%	0%	0%

Thus, AMT assists users, developers, testers, designers to check the compatibility of an Android app across a set of devices based on simply user's interactions, allowing still generating code (script ou input for UI testing) for postponed execution. AMT will be available under GNU Lesser General Public License version 3.

5. Conclusion

In this paper, we presented our technique for assisting developers, testers, and designers to verify the behavior of an android app through different configuration devices exploring the fragmentation problem present on Android platform. This tool was designed on top of a publish/subscriber model making possible the execution of the mirror approach on severals emulators simultaneously. One distinguished feature of our work is the use of Orthogonal Array Technique (OAT) to optimize the choice of devices as replay agents. The experiments done show that it can be useful and effective in practice. As future work, we are considering use AMT to predict critical components in method level based on the user's interactions.

Acknowledgments

The authors would like to thank the Brazilian Funding Agency – CAPES, CNPq, FAPEG and FAPESP –, and the anonymous referees for their valuable comments.

References

Android Market (2016). Android market. Available at: `http://developer.android.com/about/dashboards/index.html`.

Espresso (2016). Espresso API. Available at: `https://google.github.io/android-testing-support-library/docs/espresso/index.html`.

Genymotion (2016). Genymotion. Available at: `http://www.genymotion.com`.

Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing-and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 72–81. IEEE.

Google (2016). Google play. Available at: `https://play.google.com/store`.

Hu, C. and Neamtiu, I. (2011). Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM.

IDC (2015). International Data Corporation. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

JavaWriter (2016). Available at: `https://github.com/square/javapoet`.

Kaasila, J., Ferreira, D., Kostakos, V., and Ojala, T. (2012). Testdroid: automated remote ui testing on android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, page 28. ACM.

Kipar, D. et al. (2014). Test automation for mobile hybrid applications: using the example of the bild app for android and ios.

Knych, T. W. and Baliga, A. (2014). Android application development and testability. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pages 37–40. ACM.

Liu, C. H., Lu, C. Y., Cheng, S. J., Chang, K. Y., Hsiao, Y. C., and Chu, W. M. (2014). Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132. IEEE.

Rabbitmq (2016). Rabbitmq. Available at: `https://rabbitmq.com`.

Pharos: Uma Ferramenta para Identificação de Defeitos em Nível de Métodos a partir de Commits e Gerenciadores de Defeitos

Kenyo Abadio Crosara Faria[1], **Eduardo Noronha de Andrade Freitas**[1], **José Carlos Maldonado**[2], **Auri Marcelo Rizzo Vincenzi**[3]

[1]Departamento de Informática – Instituto Federal de Goiás (IFG)
Goiânia – GO – Brasil

[2]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (USP) – São Carlos, SP – Brasil

[3]Departamento de Computação
Universidade Federal de São Carlos (UFSCAR) – São Carlos, SP – Brasil

{efreitas,kenyo.faria}@ifg.edu.br, jcmaldon@icmc.usp.br, auri@dc.ufscar.br

Abstract. *This work presents a tool called Pharos, which aims to assist researchers and developers on identifying faulty methods (and other involved components) through a mining process on software repositories. This is possible by analyzing information that comes from both source code and issue tracking repositories. In addition to identifying involved files in commits regarding bug fixes, Pharos presents the set of methods involved in these commits and also some static metrics about affected components.*

Resumo. *Este trabalho apresenta uma ferramenta intitulada Pharos, cujo objetivo é auxiliar pesquisadores e desenvolvedores na identificação de métodos defeituosos (e outros componentes envolvidos) através de um processo de mineração em repositórios de código fonte. Isto é possível por meio do cruzamento de dados de repositórios de código fonte com dados de repositórios de rastreamento de solicitações. Além de identificar arquivos envolvidos em commits relacionados a correção de defeitos, Pharos apresenta os métodos envolvidos nestes commits e algumas métricas estáticas a cerca dos componentes afetados.*

URL para o vídeo: `https://youtu.be/LX7bireWQ5s`

1. Introdução

Compreender as causas da baixa qualidade e da geração de defeitos tem feito pesquisadores investirem recursos em modelos de predição e localização de defeitos baseado em análise estática. A existência de repositórios com defeitos reais é extremamente valiosa para validação destas pesquisas. A necessidade de explorar repositórios de código fonte em busca de defeitos motivou a prática de integração de ambientes de repositório de solicitações (por exemplo o *Bugzilla* [Serrano and Ciordia 2005]) com ambientes de controle de versão (por exemplo SVN [1]), de forma que solicitações criadas nos repositórios

[1]Repositório de código fonte e controle de versões

de defeitos pudessem ser identificadas nas operações de *commit*. Tal prática foi explorada em trabalhos como [Śliwerski et al. 2005, Zimmermann et al. 2007, Moser et al. 2008, Schröter et al. 2006, Kim et al. 2006, Neuhaus et al. 2007]. Para auxiliar pesquisadores no processo de validação de pesquisas que demandem mineração em repositórios de erros, este trabalho apresenta a ferramenta Pharos, capaz de identificar métodos afetados por *commits* relacionados à correção de defeitos, além de apresentar métricas estáticas a cerca dos artefatos envolvidos.

2. Descrição da Ferramenta

Ambientes de desenvolvimento de software geralmente fazem uso de ferramentas e táticas para auxiliar na rastreabilidade de artefatos de software. Alguns ambientes promovem formas de rastrear desde os artefatos de requisitos até artefatos de testes. Para auxiliar neste processo foram utilizadas duas das ferramentas mais amplamente utilizadas na indústria são o *Jira* [Atlassian Company 2015], ferramenta capaz de realizar rastreamento de solicitações, e o *Git* [Git SCM 2015], utilizado no controle de versões de artefatos de código. A Figura 1 ilustra como os dois ambientes podem trabalhar de forma integrada provendo rastreabilidade.

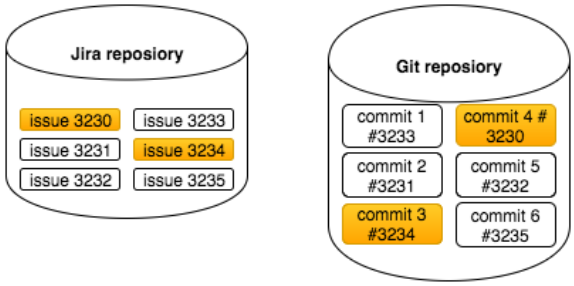


Figura 1. Repositórios Jira e Git.

O esquema apresentado pela Figura 1, mostra o repositório de código fonte contendo em seus *commits* uma referência à solicitação atendida pelo referido *commit*, no exemplo apresentado os *commits* 3 e 4, em laranja, atendem às solicitações 3234 e 3230, respectivamente. Isso é possível por meio de uma *tag* com o identificador da solicitação no corpo da mensagem do *commit*.

Com base nesta prática, a Pharos foi desenvolvida com o intuito de identificar métodos alterados em prol da resolução de defeitos, além de prover um histórico dos artefatos envolvidos desde a sua criação. Atualmente a ferramenta trabalha com arquivos escritos em *Java* e, portanto, arquivos que não possuem extensão *.java* são ignorados.

2.1. Visão geral

A identificação de métodos afetados bem como o histórico de componentes envolvidos em *commits* é possível a partir do cruzamento de solicitações relacionadas à correção de defeitos e os *commits* correspondentes. A partir deste cruzamento, é possível a realização de operações de *diff* [2] em arquivos e o processamento dos mesmos em busca dos métodos afetados bem como o aferimento de suas métricas estáticas.

[2]Operação para comparação de conteúdo de arquivos. Disponível em ambientes de controle de versão

Portanto, a Pharos é capaz de identificar os métodos afetados em correções de defeitos de projetos que possuam um ambiente de rastreamento de solicitações e um repositório de código fonte. Atualmente a ferramenta está integrada com o *Jira* [Atlassian Company 2015], como ambiente de rastreamento de solicitações, e com o *Git* [Git SCM 2015], como repositório de código fonte. Para a validação da ferramenta construída bem como a escrita deste trabalho foram utilizados projetos da Apache [Apache 2015].

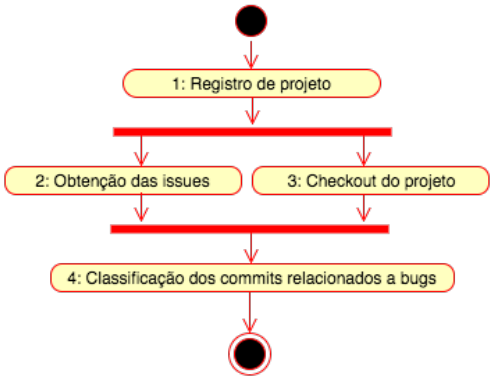


Figura 2. Diagrama de atividades básicas da Pharos

A operação da ferramenta é composta por duas fases: Fase 1 e Fase 2. A Figura 2 apresenta um diagrama de atividades inerentes à Fase 1, nesta fase o projeto é registrado a partir da tela apresentada na Figura 3. A partir desta tela o usuário pode registrar e editar um projeto (Atividade 1), bem como obter os insumos para o cruzamento das solicitações e os respectivos *commits* (Atividades 2 e 3 da Figura 2).

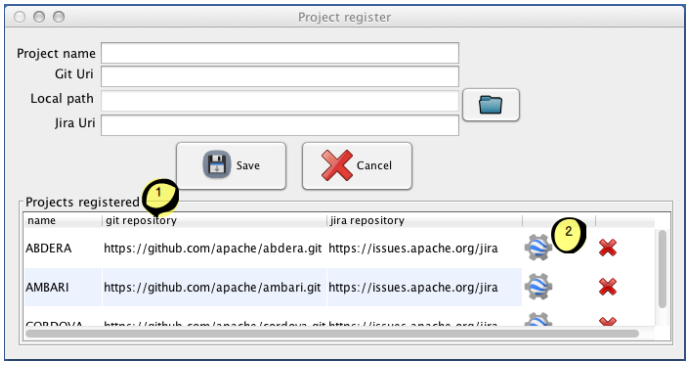


Figura 3. Tela para registro de projetos

A Atividade 4 é possível a partir da tela apresentada na Figura 4. Nesta tela, o usuário define um projeto a ser analisado, o exemplo apresentado baseia-se no software Ambari [Apache Ambari 2015], e acionando o botão marcado com o número 3 tem acesso às versões do software escolhido. Na listagem intitulada *Releases from Jira*, é possível selecionar a versão desejada e acionando o botão marcado com 4 a versão escolhida delimita o limite inferior de análise. Selecionando outra versão e acionando o botão marcado com 5 o limite superior de análise é definido. No caso do exemplo, apenas o limite superior foi definido (versão 1.2.0), portanto a análise irá considerar *commits* desde o início do projeto até a versão 1.2.0.

O acionamento do botão marcado com 6 recupera solicitações do ambiente *Jira*, estas são persistidas em um SGBD relacional durante a Atividade 1 da Figura 2. As solicitações recuperadas são apresentadas na listagem marcada com 7 (Figura 4), além da listagem, é apresentado o total de solicitações existentes no repositório bem como o total de solicitações relacionadas a defeitos.

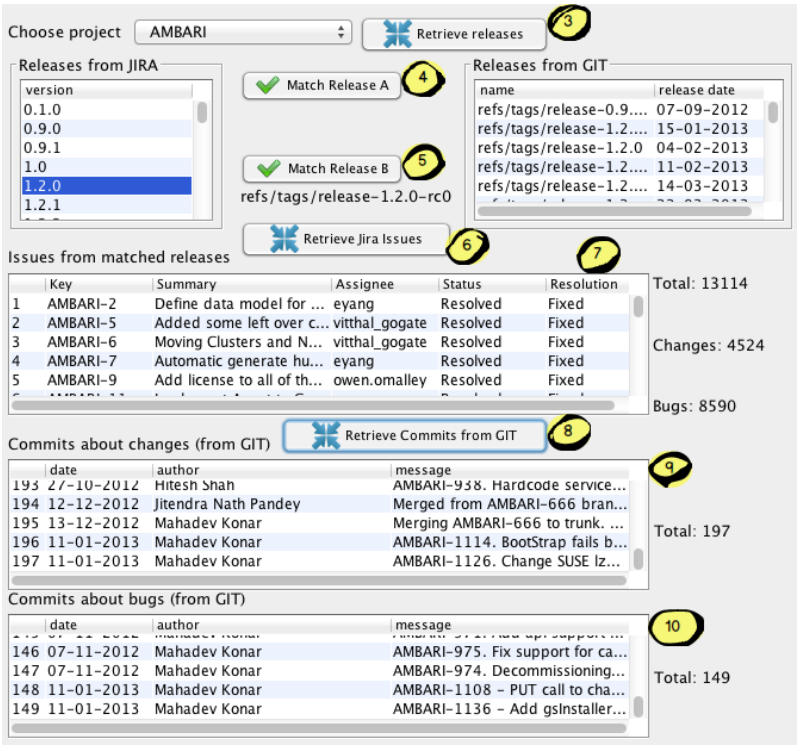


Figura 4. Visualização dos repositórios

A recuperação dos *commits* a partir do intervalo de versões definido, é possível por meio do acionamento do botão marcado com 8. Os *commits* recuperados são apresentados nas listagens marcadas com 9 e 10, na listagem 9 constam *commits* relacionados à alterações diversas e na listagem 10 constam *commits* relacinados à correção de defeitos. Considerando as versões do projeto Ambari [Apache Ambari 2015] e o intervalo de versões definido, foram encontrados 197 *commits* sem nenhuma relação com defeitos e 149 *commits* relacionados à correção de defeitos.

A Fase 2, que consiste na extração e análise dos artefatos afetados nos *commits*, inicia-se a partir da seleção de um *commit* da listagem marcada com 10, então o usuário tem acesso aos artefatos envolvidos no *commit* a partir da tela da Figura 5. Esta tela apresenta, na listagem marcada com 11, todos os arquivos afetados (modificados, adicionados ou removidos) pelo *commit*, a listagem apresenta a idade do arquivo, o número de alterações já realizadas no arquivo até a data do *commit*, o número de alterações relacionadas à correção de defeitos no arquivo até a data do *commit*, o número de linhas de código (*loc*) e o número de classes definidas no arquivo, uma vez que em um mesmo arquivo (escrito em java) é possível a definição de várias classes.

Uma vez selecionado um arquivo da listagem marcada com 11, um histórico de todos os *commits* (do intervalo de versões definido) envolvendo o arquivo selecionado é

apresentado na listagem 12. Considerando o projeto Ambari [Apache Ambari 2015], o arquivo selecionado possui 3 commits no total, desde a sua criação. O primeiro registro da listagem apresenta a alteração que deu origem ao arquivo (por isso a coluna *change type* possui o valor *add*), os demais registros indicam operações de modificação no referido arquivo. Observando a coluna *age* todos os registros possuem o valor 1, isto se deve ao fato do arquivo não possuir *commits* em diferentes versões, considerando o intervalo de versões em análise. A coluna *changes* apresenta, de forma cumulativa, o número de alterações envolvidas no arquivo até o momento do *commit*, portanto, nos registros 2 e 3, este campo permanece inalterado, uma vez que a modificação realizada entre estes *commits* foi relacionada à correção de defeito, então o campo *bugs* assume o valor 1 no registro 3. Desta forma temos 1 *commit* no registro 1 (sendo 1 para *changes* e 0 para *bugs*), 2 *commits* no registro 2 (sendo 2 para *changes* e 0 para *bugs*) e 3 *commits* no registro 3 (sendo 2 para *changes* e 1 para *bugs*).

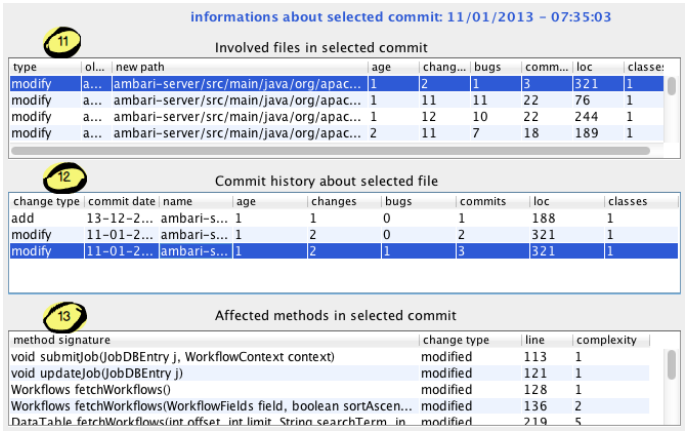


Figura 5. Visualizador de commits

O acesso aos métodos afetados em cada operação de *commit* é apresentada na listagem marcada com 13, a partir do clique em um dos *commits* da listagem 12. No exemplo do projeto Ambari [Apache Ambari 2015], o último *commit* do histórico quando selecionado, apresenta a relação dos métodos alterados, a cerca dos métodos são apresentados: a assinatura, o tipo da mudança, a linha em que inicia-se o método e a complexidade ciclomática do mesmo.

Desta forma, a Pharos identifica e apresenta todos os métodos que foram alterados em prol da correção de defeitos, permitindo que pesquisadores e desenvolvedores possam utilizar os dados para verificações de resultados de experimentos realizados com base em histórico de defeitos. Nas próximas seções são apresentados detalhes a cerca da obtenção dos dados e da arquitetura da Pharos.

2.2. Identificação dos métodos afetados

A identificação dos métodos afetados na correção de um defeito ocorre por meio da API *google-diff-match-patch* [Fraser 2012] utilizada pelo componente *phd-distiller* apresentado na Figura 6. Esta API permite a realização de operações de *diff* sem a necessidade de escrita dos arquivos em disco, apenas comparando conteúdos carregados em memória. Com base no retorno da API e o uso de expressões regulares, é possível a identificação dos métodos afetados nas alterações identificadas.

2.3. Extração das métricas

A Pharos apresenta algumas métricas sugestivas de defeitos, dentre elas as métricas *loc* (em nível de arquivo) e a complexidade ciclomática (em nível de método), estas métricas são obtidas a partir da API *checkstyle* [Burn 2003] adaptada pelo componente *phd-utilities* apresentado na Figura 6. Além destas métricas, a Pharos apresenta a idade do arquivo, pois esta é apresentada em [Catal and Diri 2009, Ostrand et al. 2005] como sugestiva a defeitos. Seu cálculo consiste do número de versões anteriores a um *commit* específico, em que um dado arquivo aparece, ou seja, se estiver em avaliação um *commit* ocorrido em 30 de dezembro de 2015, e um arquivo intitulado *parse.java* foi afetado por este *commit*, a idade deste arquivo é igual ao número de ocorrências do mesmo em diferentes versões anteriores ao *commit* avaliado, se o arquivo sofreu 10 modificações em 3 diferentes versões, sua idade é 3.

2.4. Arquitetura da Ferramenta

A Pharos foi construída para funcionar em ambientes desktop, escrito em Java e compilado para funcionar em máquinas virtuais em versões iguais ou superiores a 1.7, faz uso de um SGBD relacional (MySQL 5) para armazenamento dos projetos e suas respectivas solicitações. Após concluída esta ferramenta será disponibilizada sob licença LGPL V.3.0 [GNU 2016]. A Figura 6 apresenta um diagrama que mostra todos os componentes necessários para o funcionamento da Pharos.

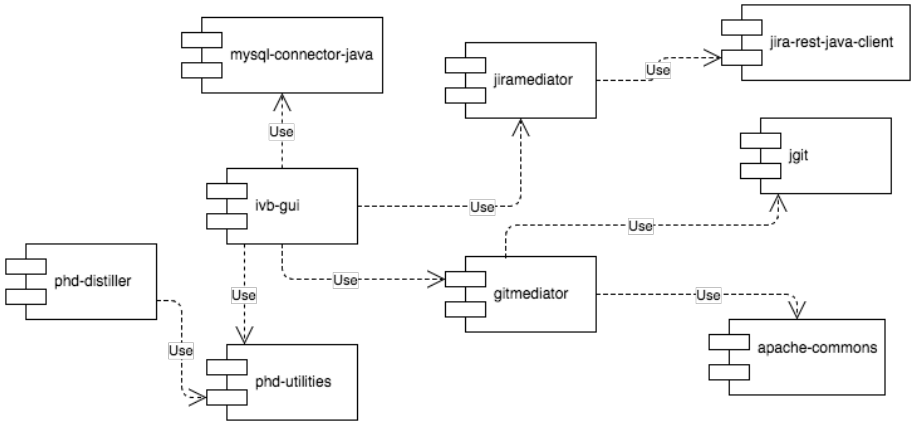


Figura 6. Componentes da Pharos

O componente *ivb-gui* aglutina artefatos como telas, componentes customizados, ordenadores de listas, formatadores e controladores. O componente *phd-utilities* aglutina funcionalidades ligadas ao processamento de código, de forma que os métodos e as classes existentes em um arquivo possam ser recuperados. Além de funções de processamento, funções como leitura e escrita de arquivos e diretórios também compõem a API deste componente. O componente *phd-distiller* consiste de um *wrapper* do componente *google-diff-match-patch* para que o uso da API seja mais intuitivo. O componente *jira-mediator* consiste de uma API que acessa repositórios *Jira* e obtem solicitações a cerca de um projeto, para isso o componente *jira-rest-java-client* é utilizado. Similarmente o componente *gitmediator* tem a responsabilidade de acessar repositórios *Git* e realizar operações básicas de qualquer cliente *Git*, isto é possível a partir do componente *jgit*. Os componentes *mysql-connector-java* e *apache-commons* são utilizados em operações com bancos de dados e arquivos.

3. Ameaças de validação

Mesmo que ambientes de desenvolvimento façam uso da prática de identificar solicitações atendidas no momento da realização de um *commit*, problemas podem ocorrer. O desenvolvedor pode não realizar a identificação da solicitação, uma vez que esta é feita na mensagem do *commit*, além disso o desenvolvedor pode se confundir e associar o *commit* a uma solicitação indevida, desta forma inconsistências podem ocorrer e prejudicar a análise.

4. Trabalhos correlatos

Algumas ferramentas foram desenvolvidas para auxiliar no processo de validação de experimentos, o LINKSTER [Bird et al. 2010] tem a proposta de permitir que o pesquisador possa marcar manualmente a motivação do *commit*, com base em uma mineração realizada em repositórios de solicitações e de código fonte, além de caixas de e-mail. [Williams and Hollingsworth 2005] apresenta uma técnica para mineração de defeitos em repositórios de código fonte mas não de forma integrada a um ambiente de rastreamento de solicitações. Portanto, nenhuma ferramenta foi encontrada com a proposta de focar na inspeção de defeitos e evolução dos artefatos envolvidos, integrando ambientes de rastreamento de solicitações e repositórios de código fonte.

5. Conclusão

A Pharos é capaz de identificar métodos afetados em correções de defeitos de forma integrada e consistente, apesar de estar suscetível a erros dos desenvolvedores no momento do *commit* os resultados alcançados são muito proveitosos à comunidade acadêmica, especialmente em tarefas de validação de modelos de predição e mineração de repositórios em busca de defeitos. Algumas melhorias são importantes tais como a disposição da evolução de métricas utilizando gráficos para comparação, inclusão de novas métricas estáticas para o enriquecimento da análise dos artefatos afetados e integração com outros repositórios de código fonte e rastreamento de solicitações, além do ranqueamento dos métodos que mais receberam correção de defeitos.

Agradecimentos

Os autores agradecem as agências de fomento CAPES, CNPq, FAPESP e FAPEG pelo apoio a essa pesquisa.

Referências

[Apache 2015] Apache (2015). Apache. Project Web Page. Available at: `http://apache.org/`. Accessed on: 01/20/2015.

[Apache Ambari 2015] Apache Ambari (2015). Ambari. Project Web Page. Available at: `https://ambari.apache.org/`. Accessed on: 01/20/2015.

[Atlassian Company 2015] Atlassian Company (2015). Jira software. Project Web Page. Available at: `https://www.atlassian.com/software/jira`. Accessed on: 01/20/2015.

[Bird et al. 2010] Bird, C., Bachmann, A., Rahman, F., and Bernstein, A. (2010). Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370. ACM.

[Burn 2003] Burn, O. (2003). Checkstyle. *SourceForge. net. Posted at http://checkstyle. sourceforge. net/(accessed October 16, 2003)*.

[Catal and Diri 2009] Catal, C. and Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058.

[Fraser 2012] Fraser, N. (2012). google-diff-match-patch-diff, match and patch libraries for plain text.

[Git SCM 2015] Git SCM (2015). Git. Project Web Page. Available at: `https://git-scm.com/`. Accessed on: 01/20/2015.

[GNU 2016] GNU (2016). Gnu. Project Web Page. Available at: `http://www.gnu.org/licenses/lgpl-3.0.en.html`. Accessed on: 01/05/2016.

[Kim et al. 2006] Kim, S., Pan, K., and Whitehead Jr, E. (2006). Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45. ACM.

[Moser et al. 2008] Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE.

[Neuhaus et al. 2007] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. (2007). Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM.

[Ostrand et al. 2005] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355.

[Schröter et al. 2006] Schröter, A., Zimmermann, T., and Zeller, A. (2006). Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 18–27. ACM.

[Serrano and Ciordia 2005] Serrano, N. and Ciordia, I. (2005). Bugzilla, itracker, and other bug trackers. *Software, IEEE*, 22(2):11–13.

[Śliwerski et al. 2005] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM.

[Williams and Hollingsworth 2005] Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466–480.

[Zimmermann et al. 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE.

JAVALI: Uma Ferramenta para Análise de Popularidade de APIs Java

Aline Brito, André Hora, Marco Tulio Valente

[1]ASERG Group – Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – Minas Gerais – Brasil.

alinebrito@ufmg.br, {hora,mtov}@dcc.ufmg.br

Abstract. *Everyday, libraries are updated, created or removed, and so their APIs. Such changes may impact stakeholders such as APIs providers and clients. In this context, it is important for APIs providers to know about strategic information like APIs change impact on clients and popularity over competitors. On the client side, it is interesting to compare APIs in order to select the most recommended in the market. To address these challenges, we propose JAVALI, a tool to analyze the popularity of Java APIs. Our database is composed of around 260K Java projects and 131M APIs. Our Web interface provides features to view and to export the results. We also report usage examples of JAVALI to solve real world API issues.*

Resumo. *Todos os dias, bibliotecas são atualizadas, criadas ou removidas, assim como suas APIs. Essas alterações podem afetar stakeholders, como provedores de APIs e clientes. Nesse contexto, é importante para os provedores conhecer informações estratégicas, como o impacto das suas atualizações e a sua popularidade sobre os concorrentes. No lado do cliente, é interessante comparar APIs relacionadas ou concorrentes, visando detectar as mais recomendadas no mercado. Para enfrentar esses desafios, esse artigo apresenta JAVALI, uma ferramenta para analisar a popularidade de APIs Java. Utiliza-se um dataset com cerca de 260 mil projetos Java e 131 milhões de APIs. A interface Web possui recursos para visualizar/exportar os resultados. Apresenta-se também exemplos de uso da ferramenta JAVALI para resolver problemas do mundo real.*

Vídeo da ferramenta. `https://youtu.be/N_yWxmEXx9o`

1. Introdução

A popularidade de um *software* é uma informação valiosa para compreender o quão bem (ou mal) está a sua evolução. Uma solução para medir a popularidade é através da apreciação do mesmo, verificando métricas em plataformas sociais de armazenamento de código, como por exemplo, o número de estrelas e observadores no GitHub [Borges et al. 2015]. Outra solução é medir o seu uso efetivo, verificando quantos são os seus clientes. Embora seja mais fácil saber o quanto um projeto é popular pelo seu número de estrelas, saber a real frequência da sua utilização é mais desafiador, uma vez que deve-se conhecer todos os sistemas clientes possíveis.

Muitos desses sistemas são bibliotecas[1], que constantemente são atualizadas, criadas e removidas, implicando em alterações nas suas APIs (*Application Programming Interfaces*) [McDonnell et al. 2013, Hora et al. 2014]. Nesse contexto, avaliar o uso de APIs é importante tanto para seus provedores como para seus clientes, uma vez que fornece informações não disponibilizadas por métricas de apreciação. Por exemplo, para o provedor da API android.view.View, comumente utilizada para criar *widgets* em aplicativos Android, saber quantos são os seus clientes e assim o público afetado por uma atualização é uma atividade crítica. Já os clientes, normalmente estão interessados em saber se outros sistemas estão compartilhando a sua decisão de usar uma determinada API, já que existem APIs semelhantes no mercado (por exemplo, para testes unitários existem duas bibliotecas famosas, org.junit e org.testng).

Com o objetivo de auxiliar tanto provedores quanto clientes de APIs nesses desafios, esse trabalho apresenta JAVALI (JAVA *Libraries and Interfaces*), uma ferramenta para analisar a popularidade de APIs Java de acordo com o uso por sistemas clientes, disponível em: `http://java.labsoft.dcc.ufmg.br/javali`.

Para disponibilizar informações de uso de APIs, a ferramenta se vale de um *dataset* com aproximadamente 260 mil projetos Java GitHub e 131 milhões de APIs. Para caracterizar o uso das APIs, a ferramenta concentra-se em dois níveis de granularidade: bibliotecas (por exemplo, quantos clientes estão usando JUnit?) e interfaces (por exemplo, quantos clientes estão usando org.junit.Test?).

O restante desse artigo está organizado conforme descrito a seguir. A Seção 2 descreve as principais funcionalidades, a arquitetura e o *dataset* utilizado pela ferramenta proposta. Na Seção 3, são apresentados exemplos de uso, onde são analisadas algumas interfaces e bibliotecas populares em Java. Finalmente, a Seção 4 discute trabalhos relacionados e a Seção 5 conclui o trabalho.

2. JAVALI: JAVA *Libraries and Interfaces*

Através da ferramenta JAVALI pode-se descobrir as interfaces mais utilizadas em Java, e as interfaces mais populares de uma dada biblioteca, assim como comparar duas ou mais bibliotecas/interfaces. A seguir descreve-se as principais funcionalidades disponibilizadas, as características da arquitetura e do *dataset* usado por JAVALI.

2.1. Principais Funcionalidades

JAVALI possui quatro telas principais, onde os resultados das consultas são exibidos em gráficos de barras ou tabelas:

Top Interfaces e ***Top Interfaces Charts***. Nessas telas visualiza-se o *ranking* das interfaces mais usadas. O tamanho *ranking* é dinâmico, sendo incrementado pelos usuários.

Customized Rankings e ***Customized Rankings Charts***. Nessas telas pode-se descobrir a quantidade de clientes e as interfaces mais usadas, para uma ou mais bibliotecas. Por exemplo, pode-se verificar qual biblioteca para injeção de dependências é mais popular, javax.inject ou com.google.inject, e quais interfaces de com.google.inject são mais usadas.

A Figura 1 apresenta a página inicial da ferramenta JAVALI, onde visualiza-se as cinco interfaces mais populares e informações sobre o *dataset*.

[1]Nesse trabalho, o termo "biblioteca" é utilizado para designar tanto *frameworks* quanto bibliotecas.

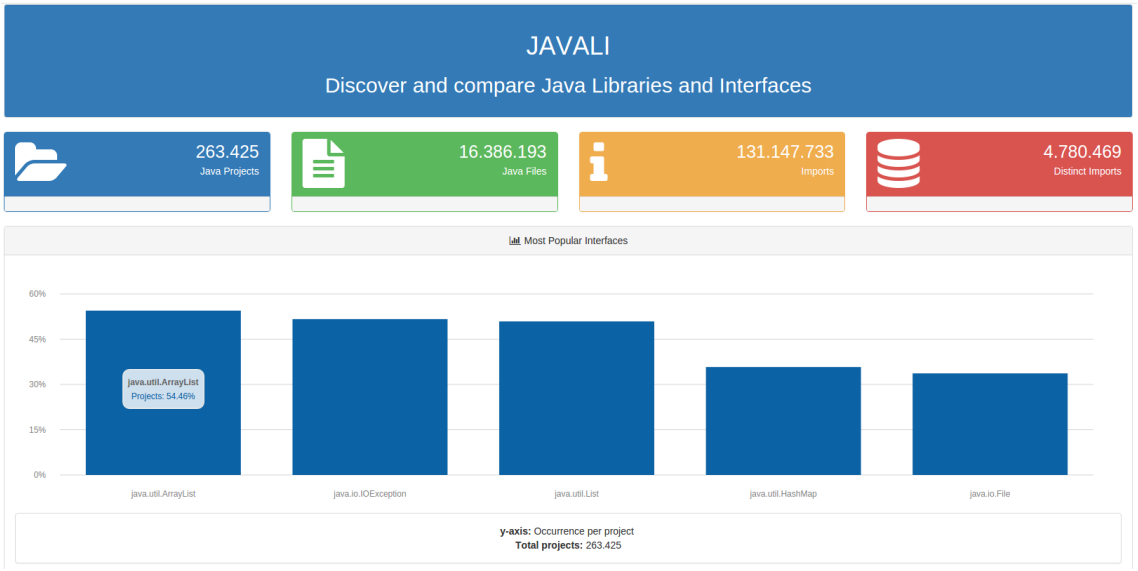


Figura 1. JAVALI – Página Inicial

2.2. Arquitetura

A Figura 2 apresenta a arquitetura de mais alto nível da ferramenta, que inclui três módulos principais: Mineração de Dados, Processamento e *Front-end*.

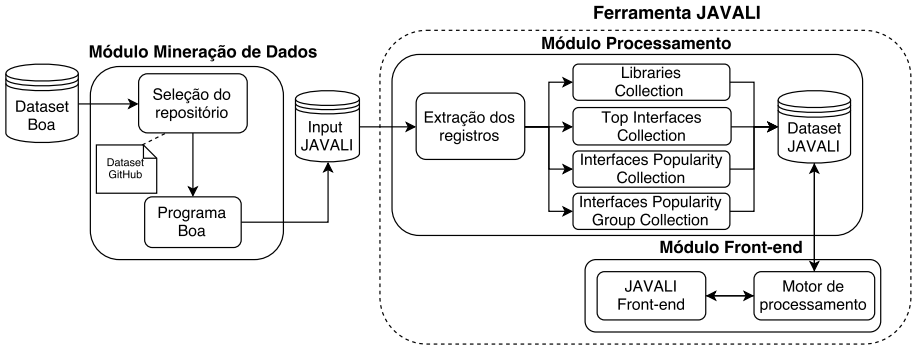


Figura 2. JAVALI – Arquitetura da Ferramenta

Mineração de Dados. Esse módulo é responsável pela obtenção dos dados da ferramenta JAVALI, minerados através da infraestrutura Boa [Dyer et al. 2013]. A Seção 2.3 apresenta mais detalhes sobre o mesmo.

Processamento. Esse módulo fornece os metadados de uso das APIs. Em **Extração dos Registros** os dados são extraídos em formato JSON, sendo organizados em quatro coleções, visando diminuir o custo computacional das consultas. Calcula-se o uso de interfaces por projeto. Assim, evita-se que interfaces muito usadas por poucos clientes sejam contabilizadas indevidamente.

Front-end. Esse módulo é responsável por interpretar as solicitações dos usuários e apresentar os resultados nas telas. Envia-se para o **Motor de Processamento** os parâmetros da consulta, onde as opções são interpretadas e a resposta retornada em formato JSON para a interface *Web*. Em **JAVALI Front-end**, os resultados são verificados, formatados

e apresentados para os usuários. Utilizou-se os *frameworks* Angular, Bootstrap e Node.js para desenvolvimento das páginas *Web* e comunicação com o banco de dados MongoDB.

2.3. *Dataset*

JAVALI provê informações a partir de um *dataset* composto por 263.425 projetos Java GitHub, 16.386.193 arquivos Java e 131.147.733 de APIs/bibliotecas. Para extrair esses dados utilizou-se a infraestrutura Boa, uma ferramenta de mineração de *software* em larga escala [Dyer et al. 2013]. A mesma possui uma linguagem própria, caracterizada por uma programação declarativa e tipos específicos de domínio para acessar os metadados dos projetos. Além de uma DSL (*Domain Specific Language*) para mineração dos repositórios de *software*, o ambiente Boa inclui uma interface *Web* para execução dos programas. Utilizou-se o dataset GitHub dessa infraestrutura, composto por 554.864 projetos Java (referente a Setembro/2015).

Para minerar os dados, criou-se um *script* na linguagem Boa para extrair os metadados de todos os projetos que possuem pelo menos um arquivo Java e que importam no mínimo uma interface/biblioteca. Após a execução desse *script*, obteve-se 263.425 projetos válidos. Os metadados extraídos foram então utilizados como entrada para a base de dados atual da ferramenta JAVALI. Adicionalmente, JAVALI aceita entradas de dados de outras fontes, desde que o formato especificado pela ferramenta seja respeitado.

3. Exemplos de Uso

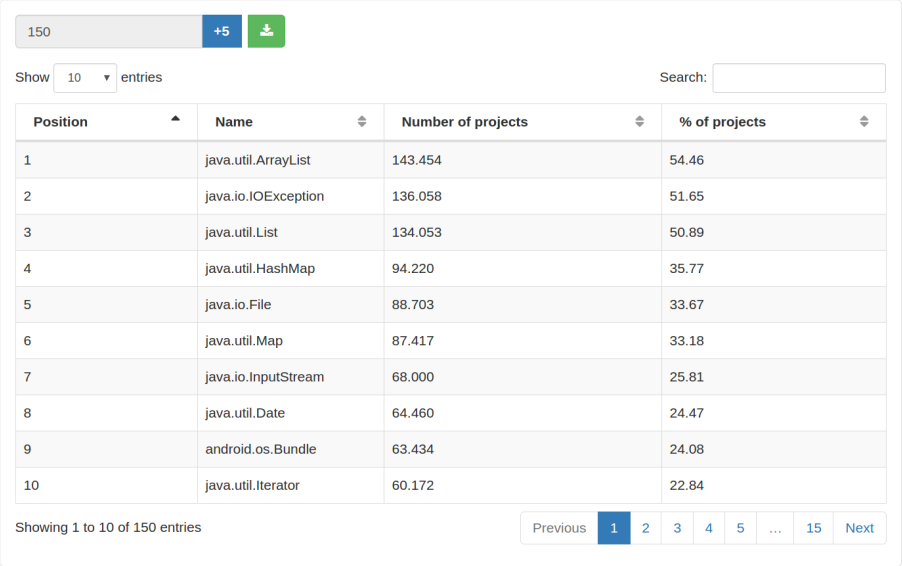
Essa seção apresenta alguns exemplos de uso da ferramenta proposta. Analisa-se no primeiro exemplo quais são as interfaces Java mais populares. No segundo exemplo, investiga-se quais interfaces estão atraindo mais clientes para as bibliotecas Android, Java, e JUnit, por serem amplamente utilizadas e bem consolidadas. No terceiro exemplo de uso, compara-se a popularidade de algumas bibliotecas e interfaces semelhantes.

3.1. Interfaces mais Utilizadas na Linguagem Java

O primeiro exemplo concentra-se na tela **Top Interfaces** do JAVALI, onde investiga-se quais são as interfaces mais usadas na linguagem Java. A Figura 3 apresenta a interface *Web* do JAVALI após essa consulta.

Esse *ranking* apresenta as interfaces mais usadas por sistemas do mundo real. Outros tipos de métricas, como por exemplo o número de estrelas, não fornecem esse tipo de informação, pois refletem apenas o apreço dos usuários pelo sistema [Hora and Valente 2015, Mileva et al. 2010]. Além disso, as estrelas informam a popularidade de bibliotecas, enquanto o *ranking* em JAVALI possui uma granularidade menor, onde pode-se descobrir o real uso em nível de interface, ou seja, pode-se visualizar as interfaces mais populares e quais são as suas respectivas bibliotecas.

Dentre as cinco APIs Java mais populares, apresentadas na Figura 1, java.util.ArrayList (55%), java.util.List (51%) e java.util.HashMap (36%) permitem criar estruturas de dados extremamente comuns. Já as APIs java.io.IOException (52%) e java.io.File (34%) são usadas em sistemas que demandam leitura e escrita em disco. As três interfaces Java mais populares são as únicas usadas em mais de 50% dos 263 mil projetos.



150 +5

Show 10 entries Search:

Position	Name	Number of projects	% of projects
1	java.util.ArrayList	143.454	54.46
2	java.io.IOException	136.058	51.65
3	java.util.List	134.053	50.89
4	java.util.HashMap	94.220	35.77
5	java.io.File	88.703	33.67
6	java.util.Map	87.417	33.18
7	java.io.InputStream	68.000	25.81
8	java.util.Date	64.460	24.47
9	android.os.Bundle	63.434	24.08
10	java.util.Iterator	60.172	22.84

Showing 1 to 10 of 150 entries Previous 1 2 3 4 5 ... 15 Next

Figura 3. JAVALI – Tela Top Interfaces

3.2. Interfaces Mais Importantes de uma Biblioteca

Nesse exemplo, analisa-se quais interfaces estão atraindo mais clientes para uma dada biblioteca. Para tanto, a tela **Customized Rankings** do JAVALI foi utilizada para consultar a popularidade das interfaces das bibliotecas JUnit, Java, e Android.

A Figura 4 apresenta a consulta realizada no JAVALI para descobrir as interfaces mais populares da biblioteca JUnit, utilizada para testes unitários. A coluna "% of projects" considera todos os projetos, assim pode-se visualizar a popularidade das interfaces dentre todos os possíveis clientes. Realizou-se esse tipo de consulta para as três bibliotecas citadas anteriormente. A biblioteca JUnit possui uma única interface muito popular, org.junit.Test (19%). Considerando o ecossistema da biblioteca, ou seja, apenas os 54.217 projetos que usam JUnit, esse valor corresponde a 92% dos seus clientes. A diferença entre ela e a próxima na lista, org.junit.Before (10%), é de 22.620 projetos, cerca de 41% dos clientes JUnit. A interface org.junit.Assert.assertEquals é a terceira mais usada na categoria.

As interfaces Java que mais atraem clientes são java.util.ArrayList (55%), java.io.IOException (52%), e java.util.List (51%), sendo as únicas usadas por mais de 50% dos clientes. As interfaces android.os.Bundle (24%) e android.app.Activity (22%) são as mais populares da biblioteca Android. Ainda nessa categoria, cerca de 18% dos clientes usam as interfaces android.view.View e android.content.Context, sendo que o pacote android.content possui duas bibliotecas entre as cinco mais usadas, android.content.Intent (16%) e android.content.Context (18%). A Figura 5 apresenta a tela **Customized Rankings Charts** em JAVALI, com o gráfico de barras das cinco interfaces Android mais populares.

Esse tipo de informação é relevante para os provedores da biblioteca, já que permite descobrir quais interfaces são mais populares dentre os seus clientes, assim como, a quantidade de clientes que serão afetados, por exemplo, pela atualização de uma determinada interface. No lado do cliente, a popularidade das interfaces numa dada biblioteca permite revelar quais interfaces são mais usadas pelos desenvolvedores.

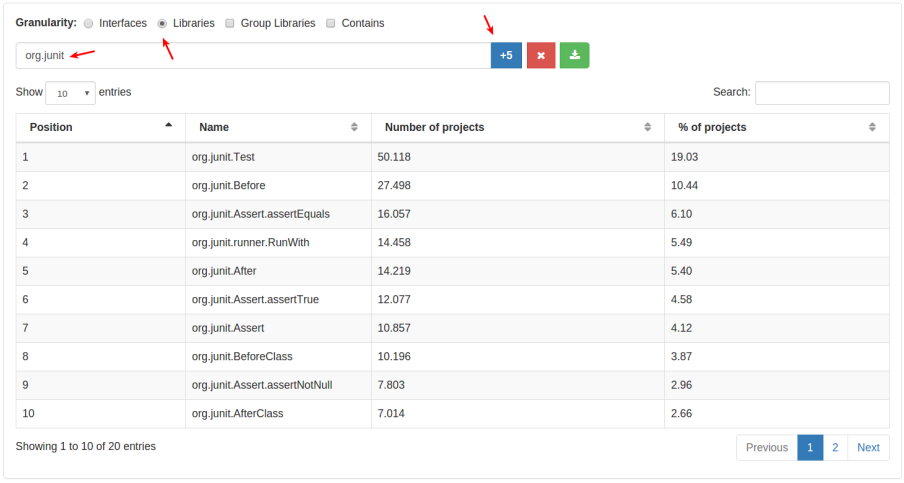


Figura 4. JAVALI – Interfaces JUnit mais Usadas

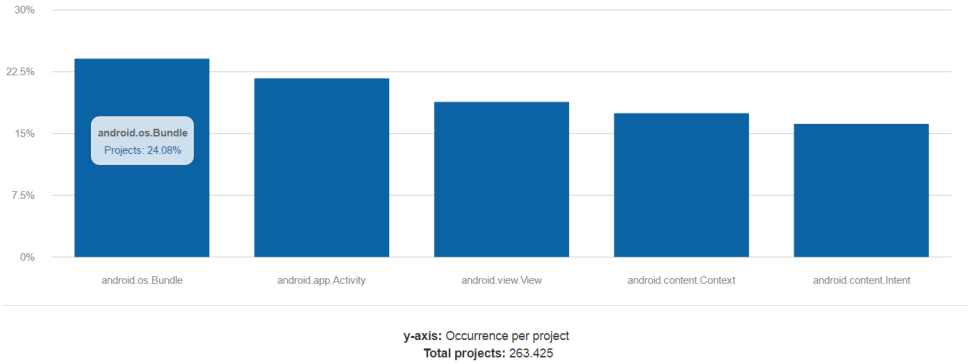


Figura 5. JAVALI – Interfaces Android mais Usadas

3.3. Comparando Bibliotecas e Interfaces

Utiliza-se nesse exemplo, a tela **Customized Rankings** do JAVALI, para comparar a popularidade de algumas interfaces e bibliotecas. Mais especificamente, compara-se bibliotecas utilizadas para realizar leituras e escritas em disco, para injetar dependências em código-fonte, para especificar testes unitários, e interfaces usadas para instanciar listas.

Escrita/Leitura em Disco. Compara-se três bibliotecas, normalmente utilizadas para realizar leitura e escrita em disco: org.apache.commons.io, com.google.common.io e java.io. A Figura 6 apresenta a consulta realizada em JAVALI para comparar as bibliotecas citadas[1]. A biblioteca mais popular é java.io (174.996 projetos - 66.43%), seguida por org.apache.commons.io (11.569 projetos - 4.39%). A terceira posição do *ranking* é ocupada pela biblioteca com.google.common.io (2.998 projetos - 1.14%).

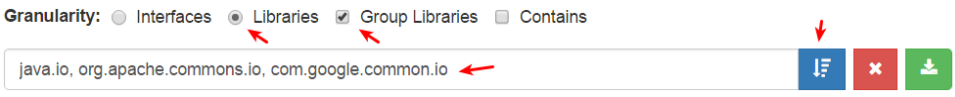


Figura 6. JAVALI – Comparar Bibliotecas para Leitura/Escrita em Disco

[1]Utilizou-se esse tipo de consulta para comparar as demais bibliotecas.

Injeção de Dependências. Para injetar dependências em código-fonte, existem duas bibliotecas famosas: com.google.inject e javax.inject.Verifica-se que javax.inject é a mais popular, usada em 6.092 projetos, enquanto com.google.inject é usada em 3.808 projetos.

Testes Unitários. Compara-se duas bibliotecas no contexto de testes unitários: org.junit e org.testng. A biblioteca JUnit é a mais popular, pois 54.217 projetos (20.58%) utilizam pelo menos uma interface JUnit, enquanto TestNG é usada em 3.588 projetos (1.36%). A diferença entre elas é de 50.629 projetos.

Instanciar Listas. Para instanciar listas, duas interfaces são comumente utilizadas: java.util.List e com.google.common.collect.Lists. A interface nativa de Java java.util.List é a mais popular, sendo usada em 134.053 projetos (50.89%), enquanto com.google.common.collect.Lists é usada por 5.120 sistemas clientes (1.94%). A Figura 7 apresenta a consulta realizada em JAVALI para compará-las.

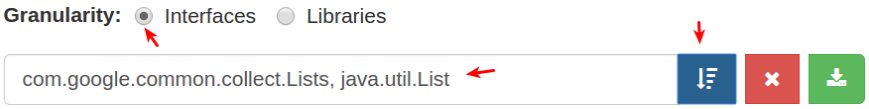


Figura 7. JAVALI – Comparar Interfaces para Instanciar Listas

A comparação da popularidade de bibliotecas e interfaces semelhantes permite aos usuários descobrirem aquelas que são mais adotadas pelo mercado. Por outro lado, os provedores podem descobrir a sua popularidade em relação aos concorrentes e o impacto de suas atualizações. Por exemplo, os provedores da biblioteca TestNG podem verificar em JAVALI que sua concorrente JUnit é mais popular, e que provavelmente a sua biblioteca precisa de intervenções para adequar-se ao mercado e atrair mais clientes.

4. Trabalhos Relacionados

Existem trabalhos que relatam a constante evolução de APIs [Hora and Valente 2015, Mileva et al. 2010]. Adicionalmente eles mencionam a necessidade de ferramentas que permitam ao cliente conhecer as APIs/bibliotecas mais adotadas no mercado, bem como os provedores conhecerem seu público alvo e abrangência das suas atualizações.

Na literatura, pode-se citar a ferramenta *apiwave*, com recursos que permitem acompanhar a popularidade de APIs e a migração dos clientes para bibliotecas concorrentes [Hora and Valente 2015]. As principais diferenças entre JAVALI e *apiwave* são: (i) JAVALI possui funcionalidades para comparar o uso de interfaces, bibliotecas, e as interfaces mais importantes por biblioteca; (ii) JAVALI disponibiliza dados sobre o uso dos pacotes das bibliotecas, ou seja, além da análise da biblioteca disponibilizada pelas ferramentas (por exemplo, a biblioteca org.springframework), pode-se explorar níveis mais internos (por exemplo, org.springframework.beans e org.springframework.stereotype); (iii) JAVALI possui a funcionalidade "*Contains*" para buscar interfaces/bibliotecas que possuem determinado pacote (por exemplo, interfaces da biblioteca org.eclipse que possuem o pacote ".internal."). Além disso, JAVALI utiliza um *dataset* composto por 263.425 projetos, enquanto *apiwave* utiliza apenas mil projetos. Para obter esse número bastante alto de projetos, JAVALI se beneficiou da infraestrutura Boa [Dyer et al. 2013].

Assim como JAVALI, a linguagem e infraestrutura Boa permite descobrir a popularidade de interfaces/bibliotecas Java [Dyer et al. 2013]. Porém é necessário o conheci-

mento prévio de uma DSL para criar o *script*, consequentemente minerar as informações de uso de interfaces e bibliotecas não é trivial. Por outro lado, JAVALI fornece uma plataforma onde torna-se simples explorar a popularidade de interfaces/bibliotecas, já que os dados são previamente processados, e a interface *Web* viabiliza a interação com o usuário.

Outro estudo no contexto de popularidade de APIs também serviu de inspiração para este trabalho [Mileva et al. 2010]. Porém, a pesquisa não propõe uma ferramenta que permita ao usuário minerar as informações de popularidade conforme a sua necessidade.

5. Considerações Finais

Este trabalho apresentou JAVALI, uma ferramenta para análise de popularidade de interfaces/bibliotecas Java. Verificou-se através de exemplos de uso quais são as interfaces Java mais populares e quais interfaces estão atraindo mais clientes para uma dada biblioteca. Além disso, comparou-se a popularidade de algumas interfaces/bibliotecas semelhantes. A partir desses exemplos procurou-se mostrar indícios de viabilidade da ferramenta para ajudar provedores e clientes a conhecerem o quão popular é uma interface/biblioteca.

Trabalhos futuros incluem uma nova versão da ferramenta, com informações de uso de bibliotecas e interfaces para outras linguagens. Todos os dados utilizados nessa pesquisa são públicos (programa Boa e entrada de dados[1], código-fonte da ferramenta JAVALI[2]) e portanto podem subsidiar futuras pesquisas nessa área.

Agradecimentos: Esta pesquisa é financiada pela FAPEMIG e pelo CNPq.

Referências

Borges, H., Valente, M. T., Hora, A., and Coelho, J. (2015). On the popularity of GitHub applications: A preliminary note. *CoRR*, abs/1507.00604.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, pages 422–431.

Hora, A., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2014). APIEvolutionMiner: Keeping API evolution under control. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Tool Demonstration Track*, pages 420–424.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of API popularity and migration. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–323.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of API stability and adoption in the android ecosystem. In *International Conference on Software Maintenance*, pages 70–79.

Mileva, Y. M., Dallmeier, V., and Zeller, A. (2010). Mining API popularity. In *International Academic and Industrial Conference on Testing - Practice and Research Techniques*, pages 173–180.

[1] `http://boa.cs.iastate.edu/boa/index.php?q=boa/job/public/35521`
[2] `https://github.com/alinebrito/JAVALI`

ArchCI: Uma Ferramenta de Verificação Arquitetural em Integração Contínua

Arthur F. Pinto[1], **Nicolas Fontes**[2], **Eduardo Guerra**[2], **Ricardo Terra**[1]

[1]Universidade Federal de Lavras (UFLA), Lavras, Brasil

[2]Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, Brasil

fparthur@posgrad.ufla.br,
{nicolas.fontes,eduardo.guerra}@inpe.br,terra@dcc.ufla.br

Abstract. *As software evolves, developers usually introduce deviations from the planned architecture, due to unawareness, conflicting requirements, technical difficulties, deadlines, etc. Although architectural compliance processes identify architectural violations, (i) these tools are underused and (ii) detected violations are rarely corrected. Therefore, this article introduces ArchCI, a tool that provides architectural compliance check as part of the Continuous Integration (CI) process. The tool relies on DCL as underlying conformance technique and Jenkins as the CI server. It implies that the architectural compliance process is triggered by every code integration, performing preset actions when violations are detected. Such actions range from sending a warning e-mail to forbid the integration to the repository. Also, this article reports case studies in a controlled and a real-world scenarios to demonstrate the applicability of the tool.*

Resumo. *No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por novos requisitos, desconhecimento, dificuldades técnicas, prazos curtos, etc. Embora existam processos e ferramentas de conformidade arquitetural que identifiquem violações, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo implementa ArchCI, uma ferramenta que provê a verificação de conformidade arquitetural como parte do processo de Integração Contínua (IC). São utilizados DCL como técnica de conformidade e Jenkins como servidor de IC. Isso implica que o processo de conformidade arquitetural é ativado a cada integração de código, executando ações configuradas quando violações forem detectadas. Tais ações podem variar desde o envio de um e-mail de alerta ao bloqueio da integração do código ao repositório. Além disso, este artigo reporta uma avaliação controlada e uma avaliação em um cenário real que demonstram a aplicabilidade da ferramenta.*

Vídeo da ferramenta. *http://youtu.be/WhjK4M--jzc*

1. Introdução

No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por desconhecimento, requisitos conflitantes, dificuldades técnicas, prazos curtos, novos requisitos, etc. [7, 11]. Isso se agrava em projetos com vários desenvolvedores uma vez que o acúmulo dos possíveis desvios arquiteturais que podem ocorrer durante sua implementação, são potencializados pelo aumento do número de desenvolvedores em um projeto, levando ao fenômeno conhecido

como erosão arquitetural [5, 2]. Mais importante, esses desvios arquiteturais impactam negativamente o projeto, podendo anular características essenciais de um sistema, como manutenibilidade, reusabilidade, escalabilidade, etc. [6]. Ainda mais crítico, como parte de um fenômeno conhecido como dívida técnica [9], sabe-se que quanto mais esses problemas demorarem para serem eliminados, mais caro será para corrigí-los.

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo apresenta ArchCI, uma ferramenta que implementa uma solução de conformidade arquitetural em Integração Contínua (IC) proposta inicialmente em uma escola [8]. Nesse cenário, o processo de conformidade arquitetural é ativado a cada integração de código no servidor – o que auxilia na solução do problema (i), pois desenvolvedores não podem desativar a ferramenta quando lhes convier –, a qual executa uma ação configurada quando violações forem detectadas, que varia do envio de um e-mail de alerta ao arquiteto de software ao bloqueio da integração ao repositório – o que auxilia na solução do problema (ii). ArchCI utiliza DCL (*Dependency Constraint Language*) como linguagem de definição das restrições de conformidade [11] e o Jenkins como servidor de IC [10].

Neste artigo, estendeu-se um estudo prévio [8] nas seguintes direções: (a) uma descrição completa das funcionalidades da ferramenta (Seção 2.1); (b) um reporte detalhado da arquitetura da ferramenta com a inclusão de vários detalhes técnicos voltados à uma sessão de ferramentas (Seção 2.2); (c) uma avaliação em um cenário real de desenvolvimento (Seção 3.2); e (d) uma revisão do estado-da-prática em relação a ferramentas relacionadas (Seção 4).

O restante desse artigo está organizado como descrito a seguir. A Seção 2 provê uma visão geral da ferramenta ArchCI, descrevendo um exemplo de uso, suas principais funcionalidades, sua arquitetura e interface. A Seção 3 avalia a aplicabilidade da ferramenta em um cenário controlado e em um cenário real. A Seção 4 discute ferramentas relacionadas e a Seção 5 apresenta as considerações finais.

2. Ferramenta ArchCI

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante de tal cenário, ArchCI garante que o processo de verificação de conformidade arquitetural seja realizado em um servidor de IC sem a necessidade de instalações em máquinas de desenvolvedores. Assim, integrações de código com violações serão sempre detectadas e devidamente tratadas, conforme ilustrado na Figura 1, onde desenvolvedores, ao integrarem código, ativam uma tarefa de verificação de conformidade arquitetural no servidor de IC que realiza ações específicas ao detectar violações arquiteturais.

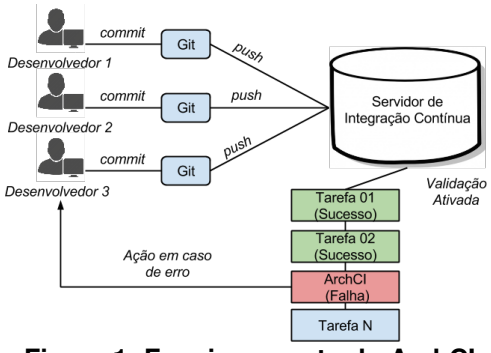


Figura 1. Funcionamento do ArchCI

ArchCI utiliza DCL como linguagem de definição das restrições de conformidade e o Jenkins como servidor de IC. ArchCI captura dois tipos de violações arquiteturais: *divergências* (quando uma dependência observada no código fonte não está de acordo com o modelo arquitetural do sistema) e *ausências* (dependência inexistente no código fonte, mas que é obrigatória de acordo com o modelo arquitetural). Essencialmente, esse modelo abrange qualquer forma de relação entre classes que podem ser verificadas estaticamente. Por restrições de espaço, uma descrição completa da linguagem DCL e do servidor de IC Jenkins podem ser encontrados em [11] e [10], respectivamente.

Como um exemplo de uso, suponha a especificação DCL (definida no arquivo `architecture.dcl`) ilustrada na Figura 2(a). A linha 3 restringe o módulo `Main` – composto pelas classes do pacote `project.main` (linha 1) – de acessar a classe `java.lang.Math`. O desenvolvedor ao tentar integrar ao repositório a classe `Main` (Figura 2(b)) que possui um acesso à classe `Math` (linha 5), ArchCI fornecerá uma mensagem de erro juntamente com as violações encontradas nas classes alteradas da integração, conforme ilustrado na Figura 2(c).

```
1 module Main: project.main.*
2
3 Main cannot-depend java.lang.Math
```

(a) Exemplo de Restrição de Dependência

```
1  package project.main;
2
3  public class Main {
4      public static void main(String[] args) {
5          System.out.println(Math.pow(2, 5));
6      }
7  }
```

(b) Exemplo de Violação

```
$ git add .
$ git commit -m "Integration Changes"
[dev a57ceb7] Integration Changes
 1 file changed, 4 insertions(+), 4 deletions(-)
$ git push

FAILURE
VIOLATION 1: 'project.main.Main' contains the method 'main' that statically invo
kes the method 'pow' of an object of 'java.lang.Math'
```

(c) Relatório após a tentativa de integração de código

Figura 2. Exemplo de Uso da ferramenta ArchCI

2.1. Funcionalidades

Verificação Arquitetural: As integrações são realizadas por intermédio do Gerrit[1] em um *branch* temporário. Quando ArchCI não detectar violações, o código é automaticamente unificado (*merge*) ao *branch* principal. No entanto, caso violações sejam detectadas, a integração ficará pendente de aprovação e ArchCI realizará, por meio de *hooks*, a ação configurada em caso de violação.

Ações: É possível (i) bloquear ou (ii) permitir a integração de código. No último caso, um *hook* do Gerrit ativará uma tarefa do Jenkins que enviará automaticamente alertas diários por e-mail ao desenvolvedor. Considerando que débitos técnicos são inevitáveis, por prazo e cobranças, cabe ao encarregado do projeto decidir qual ação corretiva é a mais adequada no projeto. Inclusive, a ferramenta pode ser desativada em atividades de reestruturação ou evolução da arquitetura.

Atomicidade: Na configuração de *bloqueio*, somente as integrações que estejam em total acordo com as regras arquiteturais do projeto serão aceitas pelo servidor.

[1] http://gerrithub.io

Verificação Incremental: ArchCI verifica somente as classes que sofreram alterações desde a última integração de forma a assegurar o bom desempenho da ferramenta.

Evolução da Arquitetura: A verificação arquitetural considera a especificação DCL armazenada no repositório (`architecture.dcl`). No entanto, em caso de uma integração em que houve modificações no arquivo `architecture.dcl` – para inclusão ou alteração de regras – ArchCI considera a especificação DCL a ser integrada ao repositório.

Uso local: ArchCI realiza a verificação de conformidade apenas no momento de integração de código. No entanto, para assegurar um menor número de violações em tentativas de integrações ao repositório, a ferramenta `dclcheck` [11] – um *plug-in* para a IDE Eclipse com a mesma finalidade – pode ser utilizada localmente.

Linguagem de Programação: Embora o conceito seja desacoplado a linguagens de programação, a atual implementação de ArchCI atua sobre projetos desenvolvidos em Java.

2.2. Estrutura Interna

Conforme ilustrado na Figura 3, a implementação de ArchCI segue uma arquitetura com cinco módulos principais:

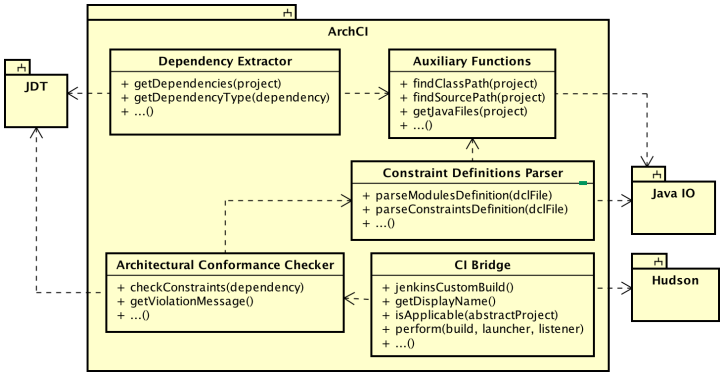


Figura 3. Arquitetura do ArchCI

Dependency Extractor: Módulo responsável pela obtenção das dependências do projeto, assim como a manipulação das mesmas. Apresenta funções que analisam cada elemento das classes a serem validadas, analisando o tipo de dependência ao qual o determinado elemento se refere. Para sua implementação, foi realizada uma adaptação da ferramenta `dclcheck` [11]. Desse modo, tornou-se necessário remover todas as dependências e partes de código que eram inteiramente exclusivos da IDE Eclipse, permanecendo apenas dependências às bibliotecas de manipulação de AST (*Abstract Syntax Tree*) fornecidas pelo Eclipse JDT (*Java Development Tools*). Todas as dependências ao Java Model, conjunto de classes que representam um projeto internamente na IDE Eclipse, tiveram de ser totalmente descartadas e, sendo assim, praticamente toda manipulação das classes a serem verificadas teve de ser reescrita. Por fim, foram utilizados métodos e funções da classe AST para que cada elemento pudesse ser compreendido e assim, determinadas as dependências do projeto.

Constraint Definitions Parser: Módulo encarregado do *parse* do arquivo contendo os módulos do projeto e as restrições de dependência estabelecidas para a arquitetura do sistema armazenadas no arquivo `architecture.dcl`.

Architectural Conformance Checker: Módulo envolvendo funções para garantir a conformidade arquitetural do projeto por meio da verificação e validação de desvios arquiteturais com base nas restrições de dependência definidas. Cada dependência extraída

pelo módulo *Dependency Extractor* é verificada frente às restrições obtidas pelo módulo *Constraint Definitions Parser* a fim de se encontrar violações arquiteturais.

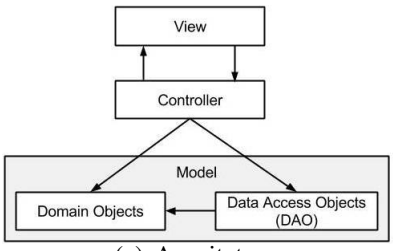
Auxiliary Functions: Módulo responsável por fornecer funções que auxiliem as tarefas do ArchCI de modo geral, por exemplo, de localização do caminho das bibliotecas e dos arquivos necessários para a resolução das dependências.

CI Bridge: Módulo contendo as funções necessárias para integrar o código ao servidor Jenkins, o qual engloba funções para a customização do *build*, obtenção do *workspace* com o código a ser integrado, identificação das classes a serem validadas, etc. O projeto da ferramenta ArchCI foi estruturado como um projeto Maven para funcionar como *plug-in* no Jenkins. Em seguida, foram designadas dependências à classes da biblioteca Hudson, para que assim fosse possível manipular os elementos e componentes envolvidos na execução das tarefas no Jenkins. Por fim, foram criadas uma classe de descrição (onde as propriedades do *build* são definidas) e uma classe contendo métodos para a obtenção de informações fornecidas pela tarefa do servidor, bem como as ações realizadas pelo *build*. Assim, com o acesso ao *workspace*, o processo de verificação e validação das dependências torna-se possível pelo *plug-in*. Já o processo de *bloqueio* é executado pelo servidor de IC em conjunto com o Gerrit.

3. Avaliação

3.1. Cenário Controlado

Sistema Alvo: `myAppointments` [6], um sistema de gerenciamento de informação pessoal simples. Apesar de ser um sistema de pequeno porte, suas restrições arquiteturais são provavelmente utilizadas em diversos projetos reais. Conforme ilustrado na Figura 4(a), o sistema segue o padrão arquitetural *Model-View-Controller* (MVC). Internamente ao componente *Model*, estão contidos *Domain Objects*, que representam entidades de domínio, e *Data Access Objects* (DAOs), que encapsulam o *framework* de persistência.



(a) Arquitetura

```
%Modulos
module Controller:    myappointments.controller.*
module View:          myappointments.view.*
module Model:         myappointments.model.**
module Domain:        myappointments.model.domain.*
module Util:          myappointments.util.*
module DB:            myappointments.model.DB
module DAO:           "myappointments.model.[a-zA-Z0-9/.]*DAO"
module JavaAwtSwing:  java.awt.**, javax.swing.**
module JavaSql:       java.sql.**

%Restricoes
only View can-depend JavaAwtSwing                          %RA1
only DAO, DB can-depend JavaSql                            %RA2
View cannot-depend Model                                   %RA3
Domain can-depend-only $java                               %RA4
DAO can-depend-only Domain, Util, javaSql                  %RA5
Util cannot-depend $system                                 %RA6
```

(b) Especificação DCL

Figura 4. Avaliação Controlada com myAppointments

Restrições: `myAppointments` deve respeitar as seguintes restrições arquiteturais: (RA1) Somente *View* pode depender de *AWT/Swing*; (RA2) Somente DAOs e `model.DB` podem depender dos serviços de persistência; (RA3) A camada *View* não pode acessar componentes do *Model* diretamente; (RA4) *Domain Objects* devem depender apenas da API de Java; (RA5) DAOs podem depender somente de *Domain Objects*, do pacote *Util* e dos serviços de persistência; (RA6) O pacote *Util* não pode depender de classes específicas do projeto. Para a utilização de ArchCI, tais definições foram traduzidas para restrições de dependência na linguagem DCL, conforme ilustrado na Figura 4(b).

Violações: Como `myAppointments` foi projetado para ilustrar técnicas de conformidade, sua arquitetura original não possui violações. No entanto, para ilustrar o funcionamento

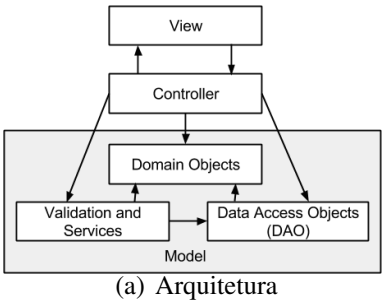
da ferramenta proposta, foram intencionalmente incorporadas seis violações arquiteturais, uma para cada restrição arquitetural. Por exemplo, a classe `Appointment`, que pertence a camada *Domain*, teve incorporadas dependências com `javax.swing.JOptionPane` e `java.sql.Date` que violam, respectivamente, as RA1 e RA2.

Resultado: Ao realizar a integração de código, ArchCI foi capaz de encontrar as seis violações com sucesso. Como a ferramenta, nesta avaliação, foi configurada de forma a não ser possível integrar código com violação arquitetural, ArchCI cancelou a integração (*push*) e informou as violações ao desenvolvedor.

Limitações: A avaliação foi realizada em um ambiente controlado – um sistema de pequeno porte, um único desenvolvedor, poucas integrações de código e um pequeno conjunto de violações. Entretanto, o objetivo da avaliação de se verificar a aplicabilidade de ArchCI foi atingido ao se demonstrar que é sim possível integrar um processo de conformidade arquitetural em IC.

3.2. Cenário Real

Sistema Alvo: LEONA - Rede Colaborativa para Estudos de Eventos Luminosos Transientes (ELT). O projeto se concentra em desenvolver uma plataforma web capaz de monitorar estações remotas localizadas em toda a América Latina e obter imagens para análise e estudos sobre os ELTs. É um sistema com uma razoável quantidade de usuários e com uma arquitetura que envolve diversos nós. O sistema é baseado no padrão arquitetural MVC, porém também possui camadas adicionais no componente Model. Existem camadas para classes de domínio, serviços com regras de negócio e de persistência, que implementam o padrão DAO. O software utiliza bibliotecas externas, sendo elas Esfinge, JavaxSwing e VRaptor, que podem ser utilizados apenas por certos componentes. A Figura 5(a) apresenta o modelo arquitetural do LEONA. É importante ressaltar que a construção do sistema já era feita dentro de um processo de IC.



(a) Arquitetura

```
%Modules
module Controller:    br.leona.server.controller.*
module Model:         br.leona.server.model.*
module Service:       br.leona.server.service*
module DAO:           br.leona.server.dao.*
module Persistence:   javax.persistence.*
module Esfinge:       org.esfinge.*
module Serializable:  java.io.Serializable.*
module VRaptor:       br.com.caelum.vraptor.*
module JavaxSwing;    javax.swing.*

%Constraints
Service can-depend-only Controller                    %RA1
only Model can-declare Persistence                    %RA2
only Controller can-depend Controller                 %RA3
only DAO can-declare Esfinge                          %RA4
View cannot-access Model                              %RA5
Model must-declare Serializable                       %RA6
only Controller can-declare VRaptor                   %RA7
only Controller can-declare JavaxSwing                %RA8
```

(b) Especificação DCL

Figura 5. Avaliação em Cenário Real com LEONA

Restrições: LEONA deve respeitar as seguintes restrições arquiteturais: (RA1) *Service* só pode depender de *Controller*; (RA2) Somente *Model* pode declarar *Persistence*; (RA3) Somente *Controller* pode depender de *Controller*; (RA4) Somente *DAO* pode declarar *Esfinge*; (RA5) *View* não pode acessar *Model*; (RA6) *Model* deve declarar *Serializable*; (RA7) Somente *Controller* pode declarar *VRaptor*; (RA8) Somente *Controller* pode declarar *JavaxSwing*. Para utilizar a ferramenta ArchCI, as definições das restrições de dependências do LEONA foram traduzidas na linguagem DCL conforme na Figura 5(b).

Violações: Como ArchCI foi implementada depois de boa parte do projeto LEONA estar desenvolvida, na sua primeira compilação foram identificadas três violações. Duas delas foram no módulo *DAO* onde estava utilizando classes do módulo *Persistence* também e a

outra no módulo *Service* que estava utilizando umas classe do *JavaxSwing*. As violações encontradas foram recusadas com base nas restrições RA4 e RA8, respectivamente.

Resultado: A ferramenta identificou as violações na integração do código à produção. Sendo assim, por intermédio da ferramenta, o ato de integrar código foi bloqueado e as violações informadas ao desenvolvedor. Deve-se destacar que a equipe encontrou soluções diferentes para conseguir atender a estrutura. Nas violações do módulo DAO foi identificado que a classe `Consultas` estava localizada incorretamente, e para corrigir foi movida para o módulo `Model`. Já na violação do módulo `Service` um diálogo de erro era instanciado, indo contra o que estava definido na arquitetura. A correção do problema foi o envio de um erro que era tratado usando o `JavaxSwing` no módulo `Controller` .

Limitações: A avaliação foi realizada em um ambiente real de desenvolvimento, onde se encontra uma equipe com três integrantes desenvolvendo e integrando os mesmos módulos de código. A ferramenta foi utilizada em uma versão do sistema com funcionalidades já concluídas, e o uso de ferramenta foi considerado apenas na última versão. Porém, o objetivo de conseguir identificar falhas na arquitetura como parte do processo de IC foi concluído com sucesso. A ferramenta continuará integrada no ambiente de desenvolvimento do projeto LEONA para que possa ser avaliado seu uso de forma contínua.

4. Ferramentas Relacionadas

Pelo menos as seguintes ferramentas podem ser utilizadas para se tratar de problemas arquiteturais juntamente com o processo de IC:

SonarQube[2] [1, 4] é uma plataforma de código aberto que provê integração com diferentes IDE's e com o servidor Jenkins para se gerenciar a qualidade do código de um projeto. Com a definição de regras arquiteturais no SonarQube, torna-se possível obter informações a respeito do software, por exemplo, cobertura de testes, métricas, matriz de dependências, aderência a boas práticas de código, etc. A partir dessas informações também é feita a quantificaçao da dívida técnica.

Checkstyle[3] utiliza métodos de manipulação de AST para inspecionar cada elemento do código e verificar sua conformidade com regras pré-definidas. Utilizando Java, é possível definir regras customizadas sobre diferentes aspectos do desenvolvimento e, posteriormente, exportá-las para aplicação em ferramentas de análise de qualidade, revisão de código, ou mesmo em servidores de IC. Um trabalho recente reportou o uso dessas regras para conformidade arquitetural [3], onde cada regra precisava ser criada a partir da criação de uma classe. O conjunto de regras desenvolvido no Checkstyle pode ser integrado ao SonarQube para análise de qualidade.

Code Climate[4] é uma plataforma para avaliação e revisão de projetos Ruby, Javascript, PHP e Python. É possível importar repositórios remotos do Github[5] e, utilizando a métrica GPA, fornecer uma nota geral sobre qualidade do projeto. A ferramenta provê dados e relatórios relacionados a complexidade, segurança, más práticas, duplicação de código, etc. Ademais, como a plataforma provê métricas a respeito dos testes, caso o projeto contenha testes arquiteturais, o Code Climate auxilia no processo de se manter um boa arquitetura de software durante o desenvolvimento e integração de código ao repositório.

[2]http://www.sonarqube.org
[3]http://checkstyle.sourceforge.net/
[4]https://codeclimate.com/
[5]https://github.com

As ferramentas encontradas apresentam funcionalidades relacionadas a avaliação da qualidade do software durante o processo de IC, porém nenhuma delas foca diretamente em um mecanismo para verificação de conformidade arquitetural. Como visto em [3], é possível criar regras personalizadas para essa finalidade, porém é possível ver que isso ainda é muito trabalhoso, necessitando a criação de uma classe para cada regra. Por fim, em relação com à `dclcheck` [11], a qual também define regras usando a linguagem DCL, o principal ponto de originalidade de ArchCI é estender DCL de forma a aliar a verificação de conformidade arquitetural como parte do processo de IC.

5. Conclusão

Este artigo descreve ArchCI, uma ferramenta de conformidade arquitetural (DCL) incorporada em um servidor de IC (Jenkins). Como principal contribuição, a ferramenta minimiza os problemas decorrentes de um processo de erosão arquitetural através de um processo de conformidade arquitetural mais rígido, e.g., integrações de código só ocorrem quando não foram detectadas violações arquiteturais.

Como trabalho futuro, pretende-se: (i) aplicar a solução proposta em projetos reais em andamento com alta taxa de integrações, a fim de avaliar sua expressividade, aplicabilidade e desempenho; inclusive a avaliação do ArchCI pelo próprio ArchCI; (ii) avaliar as características mais importantes para aceitação dos desenvolvedores, e.g., exibição de *warnings* ou bloqueio de integração de código; (iii) definir grau de severidade para cada restrição de forma a configurar ações pelo grau de severidade, e.g., bloquear a integração caso viole restrição arquitetural que afete segurança; e (iv) conduzir estudos relacionados à detecção e tratamento de violações nos diferentes momentos de integração do código.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, FAPESP, CAPES e CNPq.

Referências

[1] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning, 2013.

[2] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[3] Paulo Merson. Ultimate architecture enforcement: custom checks enforced at code-commit time. In *Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 153–160, 2013.

[4] Paulo Merson, Joseph Yoder, Eduardo Guerra, and Ademar Aguiar. Continuous inspection. In *2nd Asian Conference on Pattern Languages of Programs (AsianPLoP)*, pages 1–13, 2014.

[5] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *20th International Conference on Program Comprehension (ICPC)*, pages 3–10, 2012.

[6] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. Static architecture conformance checking: An illustrative overview. *IEEE Software*, 27(5):132–151, 2010.

[7] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[8] Arthur F. Pinto and Ricardo Terra. Processo de conformidade arquitetural em integração contínua. In *2nd Latin-American School on Software Engineering (ELA-ES)*, pages 1–12, 2015.

[9] Carolyn B. Seaman and Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[10] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, 2011.

[11] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, 2009.

UseSkill: uma ferramenta que auxilia na realização de avaliações de usabilidade em aplicações Web

Matheus Souza[1], Rafael Ribeiro[1], Pedro Almir Oliveira[2], Pedro Santos Neto[1]

[1]Universidade Federal do Piauí
Teresina - Piauí - Brasil

[2]Instituto Federal do Maranhão
Pedreiras - Maranhão - Brasil

{matheusmmcs,raffael404}@gmail.com, pedro.oliveira@ifma.edu.br

pasn@ufpi.edu.br

Abstract. *One of the most important factors for the acceptance of a Web application is its usability. The usability testing is one of the most used methods to evaluate the usability of applications, however, its performing is time consuming and requires many resources. In order to mitigate this problem, this work proposes the UseSkill, a tool that helps conducting usability evaluations both in a controlled environment as in a production environment, aiming to reduce the cost and the time to perform such evaluations. The results of an evaluation performed with the tool indicate that its use can help detecting usability problems, that although relevant, may go undetected even to usability experts.*

Resumo. *Um dos fatores mais importantes para a aceitação de uma aplicação Web é a sua usabilidade. O teste de usabilidade é um dos métodos mais utilizados para avaliar a usabilidade de aplicações, entretanto, a sua realização é demorada e demanda muitos recursos. Para amenizar esse problema, este trabalho propõe a UseSkill, uma ferramenta que auxilia a realização avaliações de usabilidade tanto em ambiente controlado quanto em ambiente de produção, visando reduzir o custo e o tempo para realizar tais avaliações. Os resultados de uma avaliação realizada com a ferramenta indicam que seu uso pode auxiliar na detecção de problemas de usabilidade, que apesar de serem relevantes, podem passar despercebidos mesmo para especialistas em usabilidade.*

Vídeo em https://www.youtube.com/watch?v=KP6QMc0QZRk

1. Introdução

Aplicações *Web* tornaram-se muito importante na vida moderna, pois nos ajudam a resolver diversos problemas do dia a dia. Esse fato proporcionou um grande crescimento dessas aplicações, aumentando a exigência dos consumidores por qualidade. Dentre os critérios usados para avaliar a qualidade de uma aplicação *Web*, a usabilidade é um dos principais, por avaliar a qualidade de interação entre o usuário e a interface da aplicação, sendo fator determinante do seu sucesso [Fernandez et al. 2011].

Um dos métodos mais populares de avaliação de usabilidade, por avaliar diretamente a interação do usuário com a aplicação, é o teste de usabilidade. Todavia, esse tipo

de teste costuma ser dispendioso, pois necessita de inúmeros recursos físicos como salas com isolamento e equipamentos para monitoramento do usuário, recursos humanos – especialistas e auxiliares – que servirão para observar e avaliar os usuários durante o teste [Mueller et al. 2009] e de tempo para a sua execução. Essas características dificultam sua realização de forma iterativa.

Um método alternativo aos testes tradicionais é o teste automatizado/semiautomatizado, realizado a partir da captura e análise da interação do usuário com a aplicação. A extração de informações relativas à usabilidade a partir de eventos da interface tem sido considerada há bastante tempo, estimulando o desenvolvimento de várias ferramentas para este propósito [Burzacca e Paternò 2013]. Com o apoio dessas ferramentas é possível obter resultados similares aos que seriam obtidos nos testes convencionais [Tullis et al. 2002].

Este trabalho tem como objetivo propor uma ferramenta que visa reduzir os custos e a complexidade de avaliações de usabilidade para sistemas *Web*. A ferramenta UseSkill[1] permite a captura da interação do usuário (*log*), de forma remota e automática, em contextos controlados (UseSkill *Control*), com a realização de tarefas pré-definidas, e em contextos de produção (UseSkill *On-The-Fly*), onde o usuário utiliza livremente o sistema em seu dia a dia. Com base nos dados capturados, a ferramenta compara as ações realizadas por usuários "experientes" e "iniciantes" no sistema, calcula métricas associadas ao uso e com isso aponta possíveis problemas de usabilidade.

2. UseSkill: uma Visão Geral

A UseSkill foi criada para auxiliar avaliações de usabilidade a partir da comparação entre padrões de uso de usuários considerados "experientes" em relação aos "iniciantes" no uso de uma aplicação. Durante uma avaliação de usabilidade, esses dois grupos de usuários são postos para realizar as mesmas tarefas e os *logs* de suas ações são capturados, formando fluxos de ações realizadas por cada usuário. Por meio da comparação desses fluxos, a ferramenta gera relatórios que podem ser usados por um especialista para a detecção de problemas de usabilidade. Essa primeira versão da ferramenta, baseada no modelo tradicional de teste, foi denominada UseSkill *Control* [Souza et al. 2015]. A Figura 1 mostra a ideia descrita.

Apesar do auxílio e da facilidade provida pela UseSkill *Control*, foi possível observar que o fato de ter que convidar usuários para participar de uma avaliação, solicitando que executassem um roteiro pré-definido, dificultava seu uso em massa. Com base nisso, foi proposto um novo módulo denominado UseSkill *On-The-Fly* (USOTF). A USOTF realiza a captura dos dados em tempo real, a partir do uso em produção de uma aplicação, sem a necessidade dos participantes realizarem roteiros pré-definidos. A partir desses dados capturados no contexto de uso real de uma aplicação, são realizadas minerações de dados que permite visualizar indícios de problemas de usabilidade.

2.1. Arquitetura

A Figura 2 apresenta um diagrama contendo os componentes presentes na ferramenta proposta. Há uma divisão em dois blocos principais, contendo um conjunto de compo-

[1]A ferramenta está disponível no endereço `http://easii.ufpi.br/useskill/`. Para acessá-la, utilize o usuário `cbsoft@useskill.com` e senha `cbsoft`.

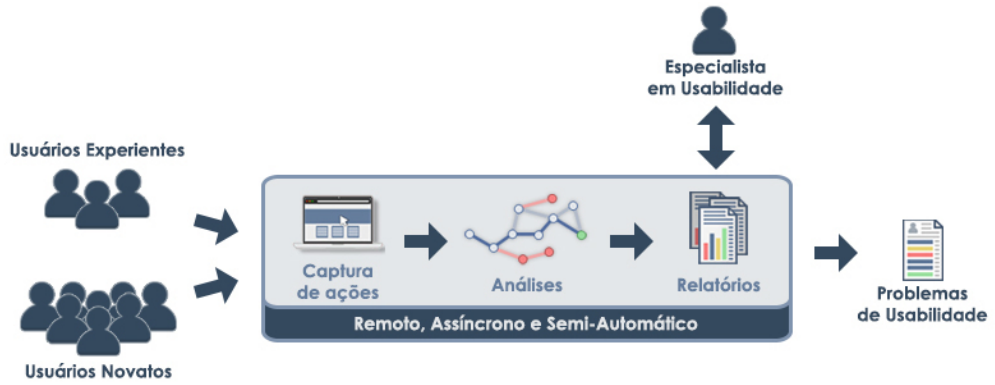


Figura 1. Abordagem da UseSkill

nentes, o bloco "UseSkill" e outro "Ferramentas Auxiliares", além de um componente desacoplado de ambos, denominado "Captura de Eventos".

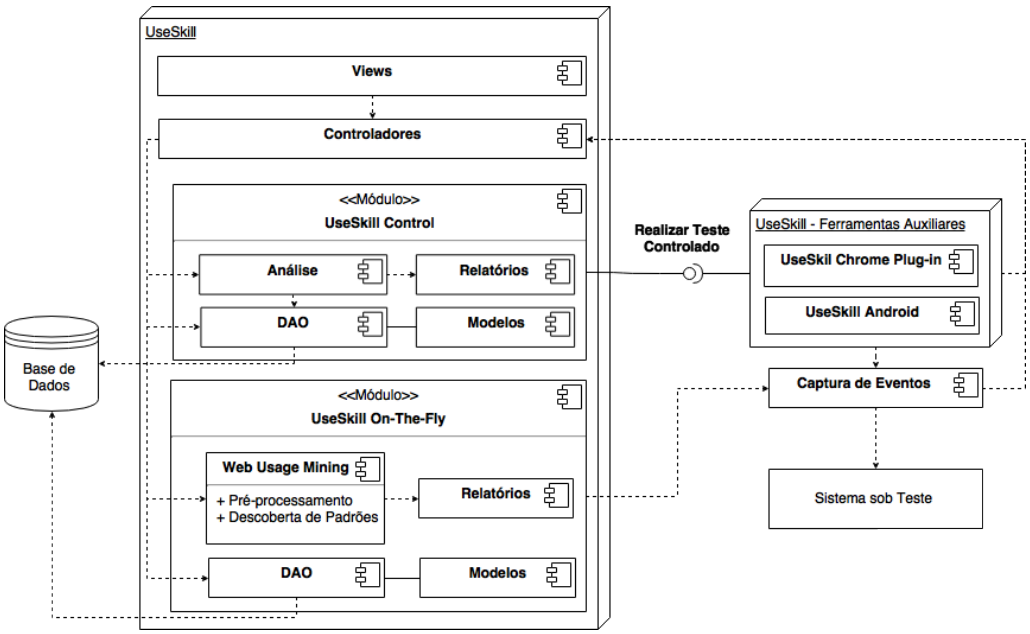


Figura 2. Diagrama de Componentes da UseSkill, agrupando módulos internos e ferramentas auxiliares

O bloco UseSkill representa a aplicação contida no servidor *Web* da ferramenta proposta. Como pode ser observado, a sua arquitetura segue o padrão de desenvolvimento MVC (*Model-View-Controller*) [Burbeck 1992], visando a separação entre a interface do usuário e a lógica do sistema. Dentro do bloco UseSkill, além das *Views* e Controladores, há dois módulos denominados UseSkill *Control* e UseSkill *On-The-Fly*. Nesses módulos são definidos os componentes responsáveis pelo Modelo, o processamento dos *logs*, geração de Relatórios e os DAOs (*Data Access Object*), que é um padrão para auxiliar a persistência de dados.

Além do grande bloco da UseSkill, há o bloco de Ferramentas Auxiliares, que contém a implementação de ferramentas que auxiliam na realização de testes controlados na USC. Essas ferramentas auxiliares realizam a inserção do componente de Captura de

Eventos no Sistema sob Teste e realizam a comunicação com os Controladores da UseSkill. O componente de Captura de Eventos também faz chamadas para os Controladores da UseSkill, mas apenas quando o teste é por meio da UseSkill *On-The-Fly*, que não possui ferramentas auxiliares.

O componente de Captura de Eventos é composto por um código em *JavaScript* que "escuta" os eventos realizados na interface. Esse *script* de captura não necessita de nenhuma configuração, podendo ser utilizado sempre o mesmo *script* para todos os testes. Apesar disso, ele também pode ser personalizado para capturar apenas ações específicas, em casos onde não se deseje avaliar todas as ações.

2.2. Funcionalidades

A ferramenta UseSkill *Control* foi implementada como uma aplicação *Web* e utiliza um *plug-in* para o *browser* como forma de se integrar ao ambiente de teste. O *plug-in* insere um *script* nas páginas testadas que captura os *logs* do usuário e os envia para o servidor da UseSkill. No caso da UseSkill *On-The-Fly*, esse *script* deve ser inserido diretamente no código fonte da aplicação testada para que a captura de dados possa ser realizada. Em ambos os casos, a inserção do *script* não exige grande esforço e leva apenas alguns instantes.

Com o módulo USC também é possível realizar a criação de testes controlados, com a definição de tarefas e perguntas, além dos usuários "iniciantes" e "experientes" que participarão do teste. Esses usuário recebem o convite em suas contas na UseSkill e podem realizar o teste a qualquer momento, e onde acharem mais conveniente. Durante o teste, a ferramenta guia os participantes, exibe as perguntas e tarefas e registra comentários sobre as tarefas, entre outras coisas.

Em ambos os módulos (USC e USOTF), a ferramenta gera grafos após a execução do teste e classifica as ações de cada fluxo percorrido pelos usuários. Também são geradas tabelas que detalham as ações realizadas em cada tarefa, além do cálculo das métricas eficiência e eficácia, poupando tempo dos especialistas na hora de analisar os relatórios, pois permite priorizar avaliações dos fluxos mais prioritários.

2.3. Exemplo de uso

Como forma de exemplificar, suponha que deseja-se realizar uma avaliação de usabilidade em uma aplicação *Web* usando como apoio a UseSkill *On-The-Fly*. O primeiro passo é a criação de um teste. O segundo passo é o cadastro das funcionalidades a serem avaliadas. Cada funcionalidade necessita da definição das ações iniciais e finais, que servem para a ferramenta identificar quando um usuário começou e concluiu a utilização de determinada funcionalidade. Para isso é necessário definir para cada ação: o tipo de ação realizada (clique, preenchimento, etc.); a URL e informações sobre o elemento da página. Também é necessário definir o tempo máximo de sessão, utilizado em casos nos quais os usuários iniciam a execução de uma funcionalidade, mas não a finalizam por algum motivo.

Além do cadastro das funcionalidades, deve ser inserido o Componente de Captura no sistema a ser avaliado. Esse componente envia as ações realizadas pelo usuário para a UseSkill, para que sejam analisados e gerados relatórios que auxiliam na análise de utilização do sistema.

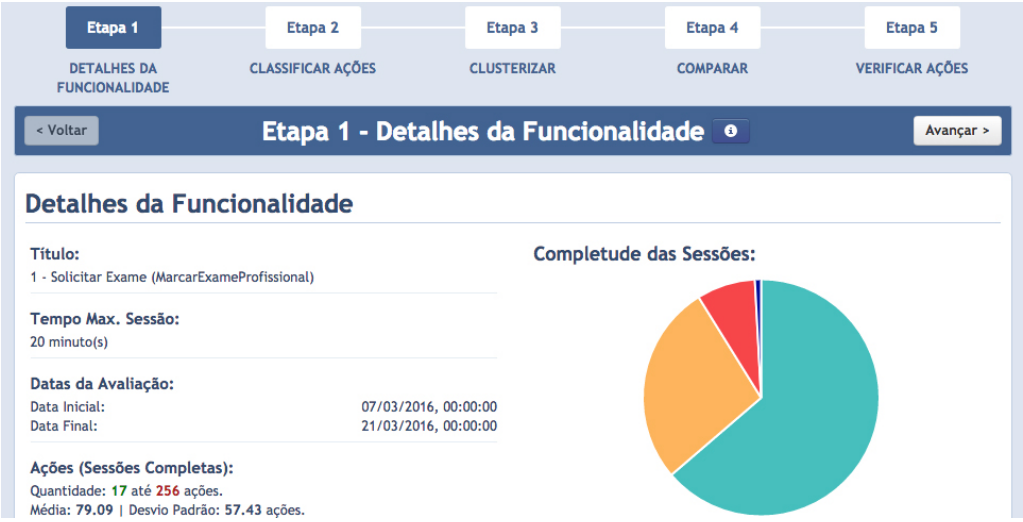


Figura 3. Interface da UseSkill durante a avaliação de uma funcionalidade.

A análise das funcionalidades é dividida em 5 etapas, vide Figura 3. A primeira Etapa permite ter uma visão geral sobre as informações analisadas pela ferramenta, como quantos usuários realizaram a funcionalidade por completo, abandonaram ou reiniciaram. A segunda Etapa apresenta o grafo de padrões sequenciais frequentes, definido a partir da mineração das sessões com o algoritmo CM-SPADE, que mostra as ações mais realizadas pelos usuários, em que é possível classificar tais ações. Nessa etapa as ações obrigatórias são informadas.

Na Etapa 3 as sessões (conjunto de ações realizadas por usuários) são agrupadas (utilizando-se o algoritmo k-means) de acordo com a qualidade de uso, criando dois grupos: um contendo as melhores sessões e outro com as demais sessões. Na Etapa 4 é feita a comparação entre os dois grupos (melhores e demais sessões), resultando na classificação de cada uma das ações de acordo com o tipo (obrigatória, problemática, correta ou alerta).

Por fim, ocorre a verificação das ações. Nela todas as sessões são listadas e podem ser ordenadas de acordo com métricas de qualidade de uso (eficácia e eficiência). Em cada sessão, o especialista consegue ver o passo a passo realizado pelo usuário em forma de grafo, a classificação de cada ação e quais ações obrigatórias não foram realizadas.

A sessão demonstrada na Figura 4 apresenta um grafo gerado pela ferramenta, onde um usuário realizou 330 ações e demorou 27 minutos e 21 segundos para concluir a funcionalidade. Nesse grafo é possível observar uma região onde os nós são maiores e as arestas mais espessas. Essas características indicam que o usuário realizou essa sequência de ações muitas vezes. Através dessa observação e da quantidade elevada de ações foi possível identificar um grande esforço para a realização da funcionalidade. Além disso, a partir de determinado ponto, o usuário realizou uma grande sequência de ações incorretas, indicando que naquele ponto (tela) específico algum problema o impediu de seguir o fluxo correto.

3. Avaliação

Foi realizado um *Quasi-experiment* com a ferramenta para tentar mensurar sua eficácia. A primeira etapa desse estudo consistiu na identificação de problemas de usabilidade em

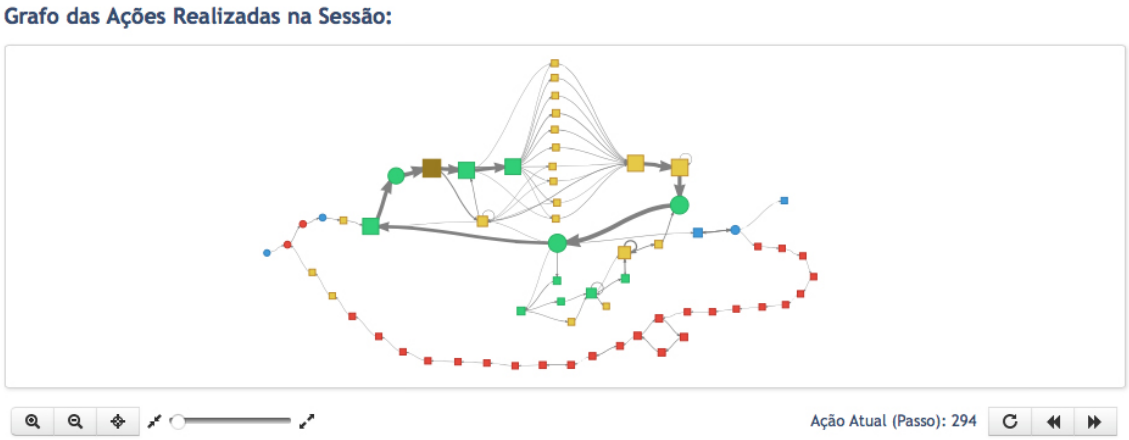


Figura 4. Grafo de sessão com eficácia 100% e eficiência 0,08%, apresentando indícios de problemas de usabilidade na funcionalidade.

seis funcionalidades de um sistema para gestão de planos de saúde. Com o apoio da UseSkill, um especialista (avaliador A) avaliou *logs* referentes a duas semanas de uso do sistema testado, contabilizando aproximadamente 410 mil ações realizadas.

Em seguida, dois *Designers* de Interfaces que trabalham na empresa que desenvolveu o sistema de gestão plano de saúde (denominados aqui de avaliador B e C) avaliaram as mesmas seis funcionalidades sem o apoio da UseSkill. Os avaliadores observaram a utilização do sistema seguindo um roteiro com seis tarefas, um para cada funcionalidade. Enquanto o sistema era utilizado, os avaliadores podiam pausar a execução para realizar perguntas sobre regras de negócio e para fazer anotações sobre os problemas encontrados.

Ao final das duas avaliações, o avaliador A, com base nos relatórios gerados pela ferramenta UseSkill, identificou 10 problemas de usabilidade. Na segunda avaliação, o avaliador B encontrou 11 problemas e o avaliador C identificou 10. Considerando que 5 problemas foram iguais entre os dois avaliadores (B e C), 16 problemas distintos foram identificados nessa segunda avaliação.

Todos os problemas de usabilidade identificados foram classificados pelos avaliadores B e C quanto a sua severidade, frequência, impacto e persistência em uma escala de zero a quatro. Quanto maior a nota, mais relevante o atributo. A Tabela 1 sumariza os resultados obtidos com as notas dadas pelos avaliadores.

Comparando as notas atribuídas aos problemas identificados com a UseSkill, os dois avaliadores consideraram que os problemas eram mais severos, impactantes e persistentes, mas que ocorrem com menor frequência em relação aos identificados por eles mesmos durante avaliação do sistema.

Como resultado do estudo realizado, percebe-se que a ferramenta foi responsável por apoiar a identificação de 7 problemas que não foram encontrados pelos avaliadores. A abordagem proposta não substitui avaliações já existentes, mas complementa com problemas que impactam diretamente na utilização do sistema. As notas atribuídas aos problemas demonstram que os problemas identificados com apoio da UseSkill são relevantes e merecem atenção. A realização da avaliação com a UseSkill não exigiu nenhum tipo de equipamento físico, tendo portanto um custo menor, e nem a organização do ambiente de

Tabela 1. Classificação dos problemas encontrados na avaliação tradicional e na avaliação com apoio da UseSkill

Avaliador e Avaliação	Problemas	Severidade Média	Frequência Média	Impacto Médio	Persistência Média
Avaliador B (Tradicional)	11	2.27	3.64	2.55	3.00
Avaliador C (Tradicional)	10	1.40	3.50	1.80	1.60
Avaliador B (UseSkill)	9	3.56	3.44	3.56	3.44
Avaliador C (UseSkill)	10	1.90	3.10	2.00	1.80

teste, sendo assim menos complexa.

4. Ferramentas Relacionadas

Foram encontrados trabalhos relacionados com propostas similares ao que foi proposto na UseSkill. Dentre eles, as ferramentas USABILICS e WELFIT merecem ênfase.

A ferramenta USABILICS realiza avaliações remotas e semiautomáticas de usabilidade de aplicações *Web*. Ao criar cada tarefa do teste com a ferramenta, são definidas as ações esperadas. Em seguida, a ferramenta compara as ações esperadas com as ações realizadas por usuários durante os testes, calculando a similaridade entre essas sequências de eventos. Para capturar os eventos dos usuários, são necessárias alterações no código fonte da aplicação a ser testada. Como resultado, a ferramenta calcula o índice de usabilidade de cada tarefa [de Vasconcelos e Baldochi Jr 2012]. Apesar das semelhanças, a UseSkill *Control* gera tabelas e grafos comparando as ações realizadas por "iniciantes" e "experientes", facilitando a identificação de possíveis problemas de usabilidade, além de não ser intrusiva.

A WELFIT [de Santana e Baranauskas 2015] é uma ferramenta que realiza a captura de *logs* automaticamente do lado do cliente, exigindo alteração do código fonte da aplicação. Durante a comparação automática dos *logs*, a ferramenta leva em consideração a distância, a partir da heurística *Sequence Alignment Method* (SAM), e o tempo médio de cada evento. Os resultados obtidos a partir da ferramenta são em forma de relatórios estatísticos e grafos dos eventos capturados. Apesar da abordagem da WELFIT ser semelhante à proposta neste trabalho, ela apresenta problemas caso haja uma grande massa de *logs*, onde a legibilidade dos grafos gerados por ela fica comprometida, dificultando a identificação pontual dos problemas de usabilidade. Para amenizar esse problema, a UseSkill permite a personalização dos eventos capturados e a configuração da visualização de grafos, além de apresentar tabelas contendo os *logs* classificados e detalhados.

5. Considerações Finais

Este trabalho apresentou a UseSkill, uma ferramenta destinada a auxiliar a realização de avaliações de usabilidade de forma remota, assíncrona e semiautomática em aplicações

Web. Muitas abordagens e ferramentas já foram propostas para a realização de avaliações de usabilidade em aplicações *Web* [Fernandez et al. 2011], o que mostra a importância de se utilizar essa técnica. Entretanto, essas ferramentas geralmente necessitam da alocação de tempo dos participantes e não avaliam o sistema em seu contexto de uso real.

A ferramenta proposta utiliza apenas *logs* para identificar possíveis problemas de usabilidade, além de poder auxiliar na realização de testes controlados, guiando os participantes nas etapas do teste, exibindo tarefas e perguntas passo a passo. Além disso, ela auxilia os especialistas, por meio da geração de relatórios de fácil interpretação. Todas essas características da ferramenta são fortes indícios de que ela pode ajudar na detecção de problemas de usabilidade.

Um estudo comparando o desempenho da UseSkill na detecção de problemas de usabilidade em relação a outro método de avaliação mostrou que a ferramenta foi capaz de encontrar problemas de usabilidade que ocorrem com menor frequência e que passaram despercebidos até mesmo aos especialistas. A ferramenta encontra-se em evolução, a partir da introdução de técnicas de mineração para apoiar ainda mais as avaliações de usabilidade.

Referências

Burbeck, S. (1992). Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2*, 5.

Burzacca, P. e Paternò, F. (2013). Remote usability evaluation of mobile web applications. In Kurosu, M., editor, *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments*, volume 8004 of *Lecture Notes in Computer Science*, pages 241–248. Springer Berlin Heidelberg.

de Santana, V. F. e Baranauskas, M. C. C. (2015). Welfit: A remote evaluation tool for identifying web usage patterns through client-side logging. *International Journal of Human-Computer Studies*, 76:40–49.

de Vasconcelos, L. G. e Baldochi Jr, L. A. (2012). Towards an automatic evaluation of web applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 709–716. ACM.

Fernandez, A., Insfran, E., e Abrahão, S. (2011). Usability evaluation methods for the web: A systematic mapping study. *Information and Software Technology*, 53(8):789 – 817. Advances in functional size measurement and effort estimation - Extended best papers.

Mueller, C., Tamir, D., Komogortsev, O., e Feldman, L. (2009). An economical approach to usability testing. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 124–129.

Souza, M. M. C., Oliveira, P. A., Ribeiro, R. F., Britto, R. S., e Neto, P. S. (2015). Useskill: uma ferramenta de apoio à avaliação de usabilidade de sistemas web. *XIV Simpósio Brasileiro de Qualidade de Software (SBQS)*.

Tullis, T., Fleischman, S., McNulty, M., Cianchette, C., e Bergel, M. (2002). An empirical comparison of lab and remote usability testing of web sites. In *Usability Professionals Association Conference*.