ANAIS PROCEEDINGS

CBSOFT * 2015

BRAZILIAN CONFERENCE ON SOFTWARE: THEORY AND PRACTICE

BELO HORIZONTE

VEM 2015

3rd WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION, AND MAINTENANCE

CBSOFT.ORG

Sponsors:

ThoughtWorks · AvenueCode · Google Brasil · take.net · RARO LABS · Avanti · IBM

Promotion:

SBC Sociedade Brasileira de Computação

Organizing Institutions:

UFMG · PUC Minas · CEFET-MG

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico · CAPES · FAPEMIG

VEM 2015

3rd WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION, AND MAINTENANCE
September 23rd, 2015
Belo Horizonte – MG, Brazil

ANAIS | *PROCEEDINGS*

COORDENADORESDO COMITÊ DE PROGRAMA DO VEM 2015 | *PROGRAM COMMITTEE CHAIRS OF VEM 2015*
Marco Aurélio Gerosa (IME-USP)
Ricardo Terra (UFLA)

COORDENADORES GERAIS DO CBSOFT 2015 |*CBSOFT 2015 GENERAL CHAIRS*
Eduardo Figueiredo (UFMG)
Fernando Quintão (UFMG)
Kecia Ferreira (CEFET-MG)
Maria Augusta Nelson (PUC-MG)

REALIZAÇÃO | *ORGANIZATION*
Universidade Federal de Minas Gerais (UFMG)
Pontíficia Universidade Católica de Minas Gerais (PUC-MG)
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

PROMOÇÃO |*PROMOTION*
Sociedade Brasileira de Computação | Brazilian Computing Society

APOIO | *SPONSORS*
CAPES, CNPq, FAPEMIG, Google, RaroLabs, Take.net,
ThoughtWorks, AvenueCode, AvantiNegócios e Tecnologia.

APRESENTAÇÃO

O III Workshop Visualização, Evolução e Manutenção de Software (VEM 2015) é um dos eventos integrantes do VI Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2015), realizado em Belo Horizonte – MG, no período de 21 a 25 de setembro de 2015. O VEM tem como objetivo integrar as comunidades das áreas de visualização, manutenção e evolução de software, oferecendo um fórum em território brasileiro onde pesquisadores, estudantes e profissionais podem apresentar seus trabalhos e trocar ideias a respeito dos princípios, práticas e inovações recentes em suas respectivas áreas de interesse. O VEM surgiu a partir da junção de dois outros workshops brasileiros focados em temas relacionados que até então vinham sendo realizados de forma separada, a saber: Workshop de Manutenção de Software Moderna (WMSWM) e Workshop Brasileiro de Visualização de Software (WBVS).

O Comitê de Programa (CP) do VEM 2015 é formado por 50 pesquisadores atuantes nas áreas de visualização, manutenção e evolução de software, provenientes de diversas regiões do Brasil e de outros países da América Latina. Os membros do CP foram responsáveis pela seleção de 15 artigos completos para serem apresentados no VEM 2015, de um total de 35 artigos submetidos. Cada artigo submetido foi avaliado por pelo menos três membros do CP, com base nos critérios de originalidade, qualidade técnica e adequação ao escopo do workshop. Os artigos selecionados abrangem diversos temas de interesse do evento, como APIs, verificadores estáticos, compreensão de programas, desenvolvimento web, ferramentas de apoio à visualização, manutenção, reengenharia e evolução de software.

Além da apresentação dos quinze artigos selecionados pelo CP, o programa técnico do VEM 2015 inclui ainda três palestras convidadas nas quais os participantes do evento poderão discutir os problemas e soluções mais relevantes atualmente nas áreas de visualização, manutenção e evolução de software, bem como novas oportunidades de pesquisa e desenvolvimento tecnológico nessas áreas.

Para finalizar, gostaríamos de agradecer a todos os autores que submeteram artigos ao VEM 2015, pelo seu interesse, aos membros do CP, pelo esforço e valiosa colaboração durante o processo de seleção dos artigos, e aos organizadores e patrocinadores do CBSoft 2015, pelo apoio na realização do evento.

Belo Horizonte, 23 de setembro de 2015

Marco Aurélio Gerosa e Ricardo Terra
Coordenadores do Comitê de Programa do VEM 2015

FOREWORD

The 3rd Workshop on Software Visualization, Evolution and Maintenance (VEM 2015) is part of the 6th Brazilian Congress on Software: Theory and Practice (CBSoft 2015), held in Belo Horizonte – MG, from September 21 to 25, 2015. Its main goal is to foster the integration of the software visualization, evolution and maintenance communities, providing a Brazilian forum where researchers, students and professionals can present their work and exchange ideas on the principles, practices and innovations related to their respective areas of interest. VEM was created from the fusion of two previous Brazilian workshops on related themes, namely Workshop on Modern Software Maintenance (WMSWM) and Brazilian Workshop on Software Visualization (WBVS).

The VEM 2015 Program Committee (PC) is composed of 50 active researchers in the areas of software visualization, evolution and maintenance, who come from several regions of Brazil as well as from other Latin American countries. The PC members selected 15 full papers to be presented at VEM 2015, from a total of 35 submissions. Each submission was evaluated by at least three PC members, based on their originality, technical quality and adequacy to the event's scope. The selected papers cover several themes of interest to the workshop, such as APIs, static checkers, program comprehension, web development, tool support for software visualization, maintenance, reengineering, and evolution.

In addition to the fifteen full papers selected by the PC, the VEM 2015 technical program also includes three invited keynote talks where the event participants could discuss the main problems and solutions related to software visualization, evolution and maintenance, as well as new research and technological development opportunities in these areas.

Finally, we would like to express our deepest gratitude to all authors who submitted their work to VEM 2015, for their interest, to the PC members, for their effort and invaluable collaboration during the paper selection process, and to the CBSoft 2015 organizers and sponsors, for their support and contribution.

Belo Horizonte, 23 September 2015

Marco Aurélio Gerosa andRicardo Terra
VEM 2015 Program Committee Co-chairs

COMITÊ DE ORGANIZAÇÃO | *ORGANIZING COMMITTEE*

CBSOFT 2015 GENERAL CHAIRS

Eduardo Figueiredo (UFMG)
Fernando Quintão(UFMG)
Kecia Ferreira (CEFET-MG)
Maria Augusta Nelson (PUC-MG)

CBSOFT 2015 LOCAL COMMITTEE

Carlos Alberto Pietrobon (PUC-MG)
Glívia Angélica Rodrigues Barbosa (CEFET-MG)
Marcelo Werneck Barbosa (PUC-MG)
Humberto Torres Marques Neto (PUC-MG)
Juliana Amaral Baroni de Carvalho (PUC-MG)

WEBSITE AND SUPPORT

Diego Lima (RaroLabs)
Paulo Meirelles (FGA-UnB/CCSL-USP)
Gustavo do Vale (UFMG)
Johnatan Oliveira (UFMG)

COMITÊ TÉCNICO | *TECHNICAL COMMITTEE*

COORDENADORES DO COMITÊ DE PROGRAMA DA TRILHA DE TRABALHOS TÉCNICOS | *TECHNICAL RESEARCH PC CHAIRS*

Marco Aurélio Gerosa(IME-USP)
Ricardo Terra(UFLA)

COMITÊ DIRETIVO | *STEERING COMMITTEE*

Cláudia Werner(COPPE/UFRJ)
Eduardo Figueiredo (UFMG)
Heitor Costa (UFLA)
Leonardo Murta (UFF)
Marco Aurélio Gerosa (IME-USP)
Nabor Mendonça (UNIFOR)
Renato Novais (IFBA)
Ricardo Terra (UFLA)

COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE*

Alexandre Bergel (University of Chile, Chile)
Aline Vasconcelos (IFF)
Auri Marcelo Rizzo Vincenzi (UFG)
Cesar Couto (IFTM)
Christina Chavez (UFBA)
Claudia Werner (COPPE-UFRJ)
Claudio Sant`Anna (UFBA)
Dalton Serey (UFCG)
Delano Beder (UFSCar)
Eduardo Figueiredo (UFMG)
Eduardo Guerra (INPE)
Elder José Cirilo (UFSJ)
Elisa Huzita (UEM)
Fernando Castor (UFPE)
Glauco Carneiro (UNIFACS)
Gustavo Rossi (Universidad Nacional de La Plata, Argentina)
Heitor Costa (UFLA)
Humberto Marques (PUC Minas)
Igor Steinmacher (UTFPR)
Ingrid Nunes (UFRGS)
João Arthur Brunet Monteiro (UFCG)
Jorge César Abrantes de Figueiredo (UFCG)
Jose Rodrigues Jr (USP)
Kecia Ferreira (CEFET-MG)
Leonardo Murta (UFF)
Leonardo Passos (University of Waterloo, Canada)
Lincoln Rocha (UFC)

Marcelo Pimenta (UFRGS)
Marcelo de Almeida Maia (UFU)
Marco Antonio Araujo (IF Sudeste-MG)
Marco Aurelio Gerosa (IME-USP)
Marco Tulio Valente (UFMG)
Marcos Chaim (USP)
Maria Cristina de Oliveira (ICMC-USP)
Maria Istela Cagnin (UFMS)
Nabor Mendonca (UNIFOR)
Nicolas Anquetil (Université Lille 1, France)
Patrick Brito (UFAL)
Paulo Meirelles (UnB)
Renato Novais (IFBA)
Ricardo Ramos (UNIVASF)
Ricardo Terra (UFLA)
Rodrigo Spínola (UNIFACS)
Rogério de Carvalho (IFF)
Rosana Braga (ICMC-USP)
Rosângela Penteado (UFSCar)
Sandra Fabbri (UFSCar)
Simone Vasconcelos (IFF)
Uirá Kulesza (UFRN)
Valter Camargo (UFSCar)

REVISORES EXTERNOS | *EXTERNAL REVIEWERS*

André de Oliveira
Bruno Santos
Fábio Petrillo
Felipe Ebert
Johnatan Oliveira
Leonardo Humberto Silva
Luciana Silva
Luiz Paulo Coelho Ferreira
Marcelo Schots
Rodrigo Souza
Vanius Zapalowski
Weslley Torres

PALESTRAS CONVIDADAS | *INVITED TALKS*

Software Visualization
Alexandre Bergel(University of Chile)

Alexandre Bergelis Assistant Professor and researcher at the University of Chile. Alexandre and his collaborators carry out research in software engineering and software quality, more specifically on code profiling, testing and data visualization.Alexandre has authored over 90 articles, published in international and peer reviewed scientific forums, including the most competitive conferences and journals in the field of software engineering. Alexandre has participated to over 85 program committees of international events. Alexandre has also a strong interest in applying his research results to industry. Several of his research prototypes have been turned into products and adopted by major companies in the semi-conductor industry and certification of critical software systems.Alexandre co-authored the book Deep Into Pharo.

Software Maintenance
Leonardo Murta(UFF)

Leonardo Murta is an Assistant Professor at the Computing Institute of Universidade Federal Fluminense (UFF). He holds a Ph.D. (2006) and a M.S. (2002) degree in Systems Engineering and Computer Science from COPPE/UFRJ, and a B.S. (1999) degree in Informatics from IM/UFRJ. He has a PQ-2 research grant from CNPq since 2009 and a Young Scientist research grant from FAPERJ since 2012. He has published over 150 papers on journals and conferences and received an ACM SIGSOFT Distinguished Paper Award at ASE 2006 and two best paper awards at SBES in 2009 and 2014, among other awards. He served in the program committee of ICSE 2014 and is the program chair of SBES 2015. His research area is software engineering, and his current research interests include configuration management, software evolution, software architecture, and provenance.

Software Evolution
Marco Tulio Valente(UFMG)

Marco Tulio Valente received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a “Researcher I-D” of the Brazilian National Research Council (CNPq). He also holds a “Researcher from Minas Gerais State” scholarship, from FAPEMIG. Valente has co-authored more than 70 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.

TRILHA DE ARTIGOS TÉCNICOS | *TECHNICAL RESEARCH TRACK PAPERS*

SESSION 1: EMPIRICAL STUDIES

SESSION 2: SOFTWARE MAINTENANCE

SESSION 3: SOFTWARE VISUALIZATION

Redocumentando APIs com Conhecimento da Multidão: um estudo de cobertura da API Swing no Stack Overflow

Fernanda M. Delfim[1], **Klérisson V. R. da Paixão**[1], **Marcelo de A. Maia**[1]

[1]LASCAM – Laboratory of Software Comprehension, Analytics and Mining
Universidade Federal de Uberlândia (UFU)

{fernanda,klerisson}@doutorado.ufu.br, marcelo.maia@ufu.br

Abstract. *API documentation, when exists, is insufficient to assist developers in their specific tasks. Few people write the API documentation for many potential readers, resulting in the lack of explanations and examples about different scenarios and perspectives. In this paper, we report a feasibility study to use the knowledge generated by the crowd on Stack Overflow for API documentation. The focus is to measure the contribution of the crowd with snippets about elements of an API in answers of how-to-do-it questions, in which the questioner provides a scenario and asks about how to implement it. We found that more than half of the Swing classes are not covered in questions-answers of that type, claiming for new incentive mechanisms for uncovered parts of the API.*

Resumo. *A documentação de API, quando existe, é insuficiente para auxiliar desenvolvedores em suas tarefas específicas. Poucas pessoas escrevem a documentação de API para muitos potenciais leitores, resultando na falta de explicações e exemplos sobre diferentes cenários e perspectivas. Neste trabalho é relatado um estudo de viabilidade em utilizar o conhecimento gerado pela multidão no Stack Overflow para a documentação de API. O foco é medir a contribuição da multidão com trechos de código sobre elementos de uma API em respostas de perguntas how-to-do-it, em que o questionador provê um cenário e pergunta sobre como implementá-lo. Foi observado que mais da metade das classes do Swing não estão cobertas em perguntas-respostas desse tipo, o que indica a necessidade de novos mecanismos de incentivo para partes não cobertas da API.*

1. Introdução

Desenvolvedores precisam de conhecimento contínuo sobre as tecnologias e ferramentas que utilizam durante a implementação de sistemas de software. A documentação dessas tecnologias e ferramentas, no entanto, normalmente não existe ou é insuficiente. Considerando API (*Application Programming Interface*), em [Robillard 2009] foi realizado um *survey* com 80 desenvolvedores da Microsoft sobre os obstáculos na aprendizagem de API. Como resultado, foi descoberto que no topo da lista de obstáculos aparecem os que são causados por recursos ausentes ou inadequados como, por exemplo, documentação.

O sucesso das mídias sociais introduziu novas maneiras de troca de conhecimento por meio da Internet [Treude et al. 2011]. Wikis, blogs e sites de perguntas e respostas (Q&A) são novas maneiras de contribuição e colaboração que têm o potencial de redefinir como os desenvolvedores aprendem, preservam e comunicam conhecimento sobre desenvolvimento de software [Parnin et al. 2012]. A informação disponível

nesse tipo de serviço é conhecida como *conhecimento da multidão* (*crowd knowledge*) [Souza et al. 2014b], em que a multidão é formada por indivíduos que contribuem com conhecimento de diferentes maneiras. No site de Q&A Stack Overflow (SO)[1], por exemplo, os indivíduos podem contribuir fazendo ou rotulando uma pergunta, fornecendo ou votando em uma resposta, entre outras ações.

O conhecimento da multidão disponível nas mídias sociais pode ser utilizado para documentar APIs. O resultado de tal documentação é chamado de *documentação pela multidão* (*crowd documentation*), isto é, conhecimento que é escrito por muitos e lido por muitos [Parnin et al. 2012][Souza et al. 2014a]. Nesse contexto, estudos sobre a viabilidade de utilizar o conhecimento da multidão para documentar APIs são necessários. Parnin et al. [Parnin et al. 2012] realizaram um estudo sobre tal viabilidade por meio da análise de cobertura de elementos de API pela multidão em threads (uma pergunta com nenhuma ou várias respostas) do SO. No entanto, nenhum critério foi estabelecido sobre os tipos de pergunta, de postagem (pergunta e resposta) e de conteúdo (texto e trecho de código, por exemplo) para a análise de cobertura. Para exemplificar a necessidade do estabelecimento de critérios para a seleção dos dados a fim de analisar a cobertura, considere as threads que possuem pergunta do tipo *Debug/Corrective* [Nasehi et al. 2012] e *Review* [Treude et al. 2011], em que são apresentados problemas e trechos de código com defeito. O conteúdo dessas threads não é necessariamente útil para a documentação de API e, portanto, não deveria ser utilizado para medir a cobertura de elementos de API.

Neste trabalho é relatado um estudo sobre a viabilidade de utilizar o conhecimento da multidão do SO para a documentação de API, onde a API Swing foi utilizada como estudo de caso por ser amplamente utilizada e bem consolidada. O principal objetivo é medir a cobertura de elementos da API para analisar como a multidão contribui para a documentação de API com *exemplos de trechos de código* de *como realizar uma determinada tarefa*. Um classificador foi utilizado para a seleção de threads com pergunta do tipo *how-to-do-it*, em que o questionador provê um cenário e pergunta sobre como implementá-lo. Exemplos de código sobre tal cenário são fornecidos nas respostas, sendo estes potencialmente úteis para a documentação de API.

O restante deste trabalho está organizado como segue. Os trabalhos relacionados são citados na Seção 2. Na Seção 3 é descrita a metodologia do estudo, que inclui os objetivos, as questões de pesquisa e a coleta e análise de dados. Os resultados do estudo são apresentados na Seção 4. Na Seção 5 são apresentadas as limitações deste trabalho, e as conclusões e os trabalhos futuros são expostos na Seção 6.

2. Trabalhos Relacionados

Nasehi et al. [Nasehi et al. 2012] definiram que os tipos de pergunta que podem ser feitas no SO podem ser descritos com base em duas dimensões: 1) o tópico da pergunta, como a principal tecnologia que a pergunta envolve, e 2) os principais interesses do questionador. Eles definiram 4 categorias sobre os interesses do questionador. Treude et al. [Treude et al. 2011] também definiram categorias sobre tais interesses, e descobriram que o SO é eficaz em revisões de código (tipo de pergunta *review*), para perguntas conceituais (*conceptual*) e para os novatos (*novice*), e os tipos de perguntas mais frequentes incluem perguntas *how-to* e perguntas sobre comportamentos inesperados (*discrepancy*).

[1] `http://www.stackoverflow.com`

A identificação automática do tipo de pergunta a partir da primeira dimensão pode ser realizada pela análise dos rótulos (*tags*) atribuídos para a pergunta. Sobre a identificação do tipo de pergunta a partir da segunda dimensão, que envolve os principais interesses do questionador, Souza et al. [Souza et al. 2014b] conduziram um estudo experimental para a classificação automática de pares de Q&A em três categorias: *how-to-do-it*, *conceptual* e *seeking-something*. Uma comparação entre diferentes algoritmos de classificação foi realizada e os melhores resultados foram obtidos com um classificador de regressão logística com taxa de sucesso global de 76.19% e 79.81% na categoria *how-to-do-it*.

Parnin e Treude [Parnin e Treude 2011] realizaram um estudo sobre como os métodos da API jQuery são documentados na Web por meio da análise dos 10 primeiros resultados retornados por pesquisas usando o Google. Eles descobriram que, exceto a documentação oficial da API, as duas fontes que mais cobrem os métodos da API são de mídias sociais: blogs de desenvolvimento de software com 87.9% e SO com 84.4% de métodos cobertos. Jiau e Yang [Jiau e Yang 2012] replicaram o estudo de Parnin e Treude para três APIs orientadas a objetos e descobriram que a cobertura de métodos pelo SO para essas APIs é consideravelmente menor do que no estudo original (84.4% para jQuery), sendo 29.11%, 59.19%, 37.64% para Swing, GWT e SWT, respectivamente.

O trabalho mais relacionado com este é o trabalho de Parnin et al. [Parnin et al. 2012], em que foi relatado um estudo empírico sobre a viabilidade de utilizar o conhecimento da multidão do SO para a documentação de três APIs (Java, Android e GWT). Para tanto, eles construíram um modelo de rastreabilidade entre elementos de API (classes) e threads do SO em que esses elementos são citados, e calcularam a porcentagem de elementos que têm ligação com alguma thread, resultando na cobertura de classes. Uma cobertura alta sugeriria que é viável utilizar o conhecimento da multidão para a documentação de API como uma fonte abrangente de conhecimento sobre os elementos de tal API. Eles descobriram que 77.3% e 87.2% das classes das APIs Java e Android, respectivamente, são cobertas pela multidão, e concluíram que apesar do potencial da documentação pela multidão em prover muitos exemplos e explicações sobre elementos de API, a multidão não é confiável para fornecer Q&A para uma API inteira. As principais diferenças entre este trabalho e o trabalho de Parnin et al. são:

Threads analisadas. Em [Parnin et al. 2012], todas as threads com pergunta rotulada com o nome da API sob investigação são utilizadas para a análise de cobertura. Neste trabalho, além disso, somente as threads com pergunta do tipo *how-to-do-it* são utilizadas.

Conteúdo de thread analisado. Em [Parnin et al. 2012], todo o conteúdo textual de postagens foi analisado. Neste trabalho, somente trechos de código foram analisados, pois o foco do trabalho é verificar a existência de exemplos de uso de elementos de API.

Tipo de postagem analisada. Em [Parnin et al. 2012], os dois tipos de postagem existentes (pergunta e resposta) foram analisados. Visto que o foco deste trabalho são threads com pergunta do tipo *how-to-do-it*, somente respostas foram analisadas.

Tipo de elemento de API analisado. Em [Parnin et al. 2012], o tipo de elemento de API analisado foi classe. Neste trabalho, classes, interfaces e enumerações foram analisadas.

3. Metodologia do Estudo

Os *objetivos* deste estudo são: 1) *analisar* discussões sobre a API Swing no SO, *com o propósito de* investigar a viabilidade de utilizar o conhecimento da multidão para documentação de API, *com respeito a* cobertura de elementos da API por trechos de código em respostas de perguntas do tipo *how-to-do-it*, e 2) *analisar* os resultados de cobertura obtidos juntamente com os resultados obtidos usando a abordagem de Parnin et al. [Parnin et al. 2012], *com o propósito de* comparar os resultados de cobertura, *com respeito aos* filtros aplicados pela abordagem proposta neste estudo. Os dois objetivos são *do ponto de vista de* pesquisadores interessados em utilizar o conhecimento provido pela multidão na documentação de APIs, *no contexto de* 38134 threads relacionadas ao Swing da versão de 21 de Janeiro de 2014 dos dados públicos do SO.

3.1. Questões de Pesquisa

A fim de alcançar os objetivos apresentados, duas questões de pesquisa foram formuladas:

RQ_1*: Qual é a proporção de elementos do Swing cobertos pela multidão no Stack Overflow em trechos de código de respostas de perguntas do tipo how-to-do-it?*

RQ_2*: Qual é a diferença de cobertura em analisar todo o conteúdo de perguntas e respostas de todas as threads e analisar somente trechos de código em respostas de perguntas do tipo how-to-do-it?*

3.2. Coleta de Dados

Os dados necessários para responder às questões de pesquisa são 1) as threads relacionadas ao Swing do SO e 2) uma lista de classes, interfaces e enumerações existentes no Swing. As threads do SO utilizadas neste estudo são da versão de 21 de Janeiro de 2014 dos dados públicos do SO, e foram baixadas do *Stack Exchange Data Dump*[2] e importadas em um banco de dados relacional. A lista de elementos existentes no Swing foi obtida pela especificação oficial da API[3], sendo 823 classes, 90 interfaces e 10 enumerações distribuídas em 18 pacotes.

3.3. Análise de Dados

A fim de analisar a cobertura de elementos do Swing em threads do SO, primeiramente foi feita a seleção das threads com rótulo "swing" por meio de correspondência exata de palavra, totalizando 38134 threads.

A análise de cobertura é realizada da seguinte maneira. Para cada thread do nosso conjunto de threads sob análise é procurado em seu conteúdo a ocorrência de citação de nome de classes, interfaces e enumerações do Swing. Quando a citação de nome de um elemento é encontrada, uma ligação entre o elemento e a thread é criada. Como resultado, cada elemento da API terá uma lista de threads em que seu nome foi citado.

A abordagem de análise de cobertura deste trabalho possui alguns filtros sobre os dados: somente threads com pergunta do tipo *how-to-do-it*, respostas como tipo de postagem e trechos de código como conteúdo de thread são analisados. Visto que um dos objetivos deste trabalho é comparar a abordagem deste com a abordagem de Parnin et

[2] `https://archive.org/details/stackexchange`
[3] `http://docs.oracle.com/javase/8/docs/api`

al. [Parnin et al. 2012], a análise de cobertura foi realizada, como explicado no parágrafo anterior, para 8 diferentes configurações: com os três filtros mencionados (abordagem deste trabalho), sem os três filtros (abordagem de Parnin et al.), e outras seis configurações com a combinação dos filtros (veja a primeira coluna da Tabela 1).

Para as análises de cobertura com configurações que utilizam somente threads com pergunta do tipo *how-to-do-it* foi utilizado o classificador de regressão logística de Souza et al. [Souza et al. 2014b] para a identificação das threads de interesse. Como o resultado do classificador são pares de Q&A, as threads foram reconstruídas em perguntas com suas respostas usando o identificador da thread. Como pode ser observado na segunda coluna da Tabela 1, 65.75% das 38134 threads relacionadas ao Swing possuem pergunta classificada como *how-to-do-it*. Para as análises de cobertura com configurações que utilizam somente trechos de código como tipo de conteúdo de thread, para cada postagem, todos os blocos de código foram coletados de dentro da *tag* `<code>`.

4. Resultados

RQ_1: Qual é a proporção de elementos do Swing cobertos pela multidão no Stack Overflow em trechos de código de respostas de perguntas do tipo how-to-do-it?

Para responder a RQ_1, a análise de cobertura foi realizada sobre as 25073 threads que possuem pergunta classificada como *how-to-do-it*, em que somente trechos de código de respostas foram analisados. Tal análise é indicada pelo número 1 na Tabela 1. Somente 51.22% das threads analisadas foram ligadas com algum elemento do Swing.

Sobre a cobertura de elementos, para 44.23% das classes, 87.78% das interfaces e 70% das enumerações do Swing existe pelo menos uma resposta de pergunta *how-to-do-it* com trecho de código no SO. Sobre as classes, 43.38% do total de classes do Swing são classes internas. Foi observado que somente 17.37% delas foram cobertas e que 64.27% das classes não cobertas são classes internas. Para interfaces e enumerações foi observado que a porcentagem de elementos internos não cobertos é ainda maior. Para interface, 33.33% das internas foram cobertas e 72.73% das interfaces não cobertas são internas. Para enumeração, 62.5% das internas foram cobertas e 100% das enumerações não cobertas são internas.

Analisando a cobertura de elementos intra-pacote usando o mapa de calor apresentado na Figura 1 foi observado que 3 pacotes (de um total de 18) foram totalmente cobertos: `javax.swing.filechooser`, `javax.swing.text.rtf` e `javax.swing.border` (representados pelos três retângulos menores em vermelho). 2 desses pacotes, no entanto, estão no top 5 pacotes com menos elementos. O pacote com mais elementos, `javax.swing`, sendo 209 classes, 25 interfaces e 7 enumerações, obteve 60.29%, 92% e 85.71% de seus elementos cobertos, respectivamente. Além disso, nenhum elemento do pacote `javax.swing.plaf.multi` (representado pelo retângulo em cinza claro) foi coberto.

RQ_2: Qual é a diferença de cobertura em analisar todo o conteúdo de perguntas e respostas de todas as threads e analisar somente trechos de código em respostas de perguntas do tipo how-to-do-it?

Para responder a RQ_2 foi realizada a análise de cobertura sem os filtros propostos neste trabalho, sendo assim o trabalho de Parnin et al. replicado (análise indicada por 2

na Tabela 1). Além disso, seis análises de cobertura complementares foram realizadas, em que os filtros foram combinados a fim de observar quais filtros contribuem mais para a diminuição de cobertura (análises indicadas de 3 a 8 na Tabela 1).

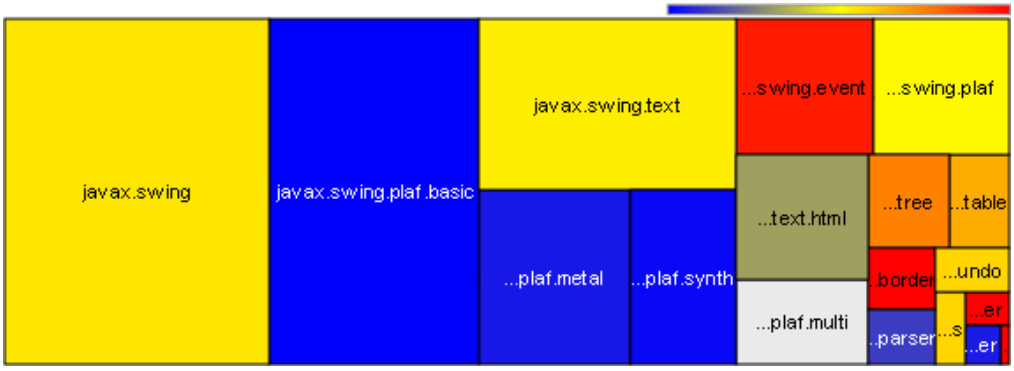


Figura 1. Mapa de calor sobre a cobertura intra-pacote, considerando classes, interfaces e enumerações. A escala de cor representa valores percentuais de cobertura sobre os elementos, sendo que azul representa o valor mais baixo e vermelho o valor mais alto. O tamanho de cada retângulo representa proporcionalmente a quantidade de elementos do pacote.

Comparando diretamente a abordagem deste trabalho com a abordagem de Parnin et al. (análises 1 e 2, respectivamente) foi observado que, apesar de que com a análise 2 mais do que o dobro de threads tenham sido ligadas com algum elemento do Swing, apenas 12.88% a mais das classes foram cobertas (57.11%), o que ainda mantém a cobertura de classes baixa. A cobertura de interfaces é razoavelmente alta na análise 1 (87.78%), e aumentou apenas 1.11% na análise 2. Para enumerações, a cobertura aumentou em 20% na análise 2, mas vale ressaltar que, observando os valores absolutos, somente 2 enumerações a mais foram cobertas.

Tabela 1. Configurações e resultados da análise de cobertura. Na primeira coluna são descritas as 8 configurações, em que três parâmetros foram combinados: tipo de pergunta (*how-to-do-it* ou todos), tipo de postagem (resposta ou Q&A) e tipo de conteúdo de thread (trecho de código ou todos).

Configuração (tipo de <>)	**Medidas sobre threads**		**Medidas sobre cobertura**		
<pergunta>, <postagem>, <conteúdo>	#Threads	#Ligadas	Classes	Interfaces	Enums
1) how-to, resposta, código	25073	12843	44.23%	87.78%	70%
2) todos, Q&A, todos	38134	27660	57.11%	88.89%	90%
3) how-to, Q&A, todos	25073	18686	53.58%	88.89%	80%
4) todos, resposta, todos	38134	27660	52.73%	88.89%	90%
5) todos, Q&A, código	38134	17993	54.07%	88.89%	80%
6) how-to, resposta, todos	25073	18686	49.45%	88.89%	80%
7) how-to, Q&A, código	25073	12843	49.70%	88.89%	70%
8) todos, resposta, código	38134	17993	47.51%	87.78%	80%

Analisando os resultados das análises de 3 a 8 foi observado que quando os filtros são aplicados separadamente (análises 3, 4 e 5) a cobertura de elementos diminui menos em comparação com as análises em que dois filtros são aplicados em conjunto (6, 7 e 8). Isso sugere que não existe uma diferença significativa em termos de cobertura para concluir que um filtro diminui mais a cobertura do que os outros, porque nenhum filtro aplicado sozinho diminui mais a cobertura do que os outros dois aplicados em conjunto.

Por exemplo, suponha que com o filtro de tipo de pergunta (somente *how-to* são analisadas) tenha sido obtida uma cobertura menor do que com os filtros de tipo de postagem (somente respostas são analisadas) e de conteúdo de thread (somente código é analisado) aplicados em conjunto. Dessa maneira seria possível concluir que o filtro de tipo de pergunta é o que mais contribui para a abordagem deste trabalho ter coberto 12.88% a menos das classes em comparação com a abordagem de Parnin et al.

Apesar de não existir uma diferença significativa em termos de cobertura para concluir que um filtro diminui mais a cobertura do que os outros, os resultados podem indicar qual filtro *não* é o que mais diminui a cobertura. Analisando os resultados das análises que combinam dois filtros (6, 7 e 8), foi observado que, para classes, as combinações do filtro *how-to* com os outros dois filtros (análises 6 e 7) produziram uma cobertura semelhante, diminuindo 4.13% de classes cobertas em comparação com a sua aplicação com nenhum outro filtro (análise 3), e para interfaces a cobertura foi mantida em 88.89%. Então, a pior cobertura para classes e interfaces é quando a análise de cobertura é feita sobre código em respostas de todas as threads (análise 8), indicando que o filtro de tipo de pergunta *how-to* não é o mais prejudicial para a diminuição de cobertura.

Adicionalmente, foi observado que com os pares de análises que variam somente o tipo de postagem – (1,7), (2,4), (3,6) e (5,8) – a mesma quantidade de threads são ligadas com elementos do Swing. Por exemplo, com as análises 1 e 7, 51.22% das threads foram ligadas com algum elemento, mesmo que a cobertura da análise 1 (com o filtro de tipo de postagem) tenha sido menor do que da análise 7 (sem o filtro). Isso sugere que as threads ligadas com o filtro são as mesmas ligadas sem o filtro, e que nas perguntas são citados elementos diferentes do que em suas respostas.

5. Limitações

Na análise de cobertura, o conteúdo de threads é analisado para a identificação de nomes de elementos de API por correspondência exata. Sendo assim, em alguns casos, citações de elementos podem ser perdidas. Por exemplo, se a classe `javax.swing.JFrame` foi citada por engano com o "f" em minúsculo, tal citação não será encontrada. Além disso, elementos que possuem o mesmo nome curto não são ligados com threads em que possuem citação somente do nome curto. No Swing, o elemento `Element`, por exemplo, foi citado em trechos de código de 66 threads, mas tais citações não foram contabilizadas para a classe `javax.swing.text.html.parser.Element` e para a interface `javax.swing.text.Element`. Ainda assim, a interface foi coberta por 9 threads em que seu nome completo foi citado, mas a classe ficou sem cobertura. Mesmo que em tal análise somente trechos de código tenham sido analisados, uma alternativa poderia ser analisar o texto da postagem na tentativa de resolver colisão de nomes.

Neste trabalho foram analisadas classes, interfaces e enumerações como tipo de elemento de API. Os resultados para esses elementos, no entanto, não podem ser generalizados para outros, como pacotes e métodos. Além disso, somente uma API (Swing) foi analisada, sendo uma limitação à generalização dos resultados para outras APIs.

6. Conclusões e Trabalhos Futuros

Neste trabalho foi apresentado um estudo sobre a cobertura de elementos da API Swing pela multidão no SO, com foco em threads com perguntas do tipo *how-to-do-it*, pois estas

são consideradas pertinentes para a documentação de API. Foi descoberto que a multidão provê trechos de código para apenas 44.23% de classes, 87.78% de interfaces e 70% de enumerações do Swing. A cobertura de classes não ultrapassou a metade da quantidade de classes existentes no Swing, e mesmo que para interfaces e enumerações a cobertura tenha sido razoavelmente alta, ainda faltam muitos elementos a serem cobertos pela multidão.

A abordagem proposta e utilizada neste trabalho, de considerar apenas trechos de código fornecidos em respostas de perguntas *how-to-do-it*, foi comparada com a abordagem de Parnin et al., em que nenhum critério sobre os dados foi estabelecido para analisar a cobertura de elementos de API. Os resultados mostraram que, com a ausência de filtros, mais do que o dobro de threads foram ligadas com algum elemento do Swing, mas apenas 12.88% a mais das classes foram cobertas, 1.11% das interfaces e 20% das enumerações (2 elementos), o que ainda mantém uma parte grande do Swing descoberta. Vale ressaltar que a quantidade de threads ligadas com elementos de API não é necessariamente importante. O importante é o conteúdo pertinente para a documentação de API, e por esse motivo neste trabalho foi proposta a utilização de threads com pergunta *how-to-do-it*.

Como trabalhos futuros, outros tipos de elementos de API, como pacotes e métodos, devem ser analisados, bem como a cobertura para outras APIs. Além disso, alternativas para chamar a atenção da multidão para discutir os elementos ainda não cobertos devem ser propostas, com o objetivo de aumentar a cobertura dos elementos de API e, consequentemente, viabilizar a documentação de API pela multidão.

Agradecimentos

Este trabalho foi parcialmente financiado pela FAPEMIG, CAPES e CNPq.

Referências

Jiau, H. C. e Yang, F.-P. (2012). Facing up to the Inequality of Crowdsourced API Documentation. *ACM SIGSOFT Software Engineering Notes*, 37(1):1–9.

Nasehi, S. M., Sillito, J., Maurer, F., e Burns, C. (2012). What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow. In *Proc. of the ICSM'2012*, pages 25–34.

Parnin, C. e Treude, C. (2011). Measuring API Documentation on the Web. In *Proc. of the Web2SE'2011*, pages 25–30.

Parnin, C., Treude, C., Grammel, L., e Storey, M.-A. (2012). Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. Technical Report GIT-CS-12-05, Georgia Institute of Technology.

Robillard, M. P. (2009). What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34.

Souza, L. B., Campos, E. C., e Maia, M. A. (2014a). On the Extraction of Cookbooks for APIs from the Crowd Knowledge. In *Proc. of the SBES'2014*, pages 21–30.

Souza, L. B. L. d., Campos, E. C., e Maia, M. A. (2014b). Ranking Crowd Knowledge to Assist Software Development. In *Proc. of the ICPC'2014*, pages 72–82.

Treude, C., Barzilay, O., e Storey, M.-A. (2011). How Do Programmers Ask and Answer Questions on the Web? In *Proc. of the ICSE'2011*, pages 804–807.

Um Estudo sobre a Utilização de Mensagens de Depreciação de APIs

Gleison Brito, André Hora, Marco Tulio Valente

[1]ASERG Group – Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – Brasil

{gleison.brito,hora,mtov}@dcc.ufmg.br

Abstract. *When an API evolves, their clients should be updated to benefit from improved interfaces. In order to facilitate this task, APIs should always be deprecated with clear replacement messages. However, in practice, there are evidences that APIs are commonly deprecated without messages. In this paper, we study a set of questions related to the adoption of API deprecation messages. We aim (i) to measure the usage of API deprecation messages and (ii) to investigate whether a tool can be built to recommend such messages. The analysis of 14 real-world software systems shows that methods are frequently deprecated without replacement messages and this problem tends to get worse with the evolution of such systems. As a result, we provide the basis for the design of a tool to support developers on detecting such missing messages.*

Resumo. *Quando uma API evolui, seus clientes devem ser atualizados para se beneficiarem de interfaces melhores. Para facilitar a atualização dos clientes, APIs devem ser sempre depreciadas com mensagens claras. No entanto, na prática, existem evidências de que APIs são comumente depreciadas sem mensagens. Nesse artigo, estuda-se um conjunto de questões relativas ao uso de mensagens de depreciação de APIs, visando (i) mensurar a utilização dessas mensagens na prática e (ii) investigar a viabilidade da construção de uma ferramenta de recomendação dessas mensagens. A análise de 14 sistemas reais confirma que métodos são frequentemente depreciados sem mensagens, e esses números tendem a aumentar com a evolução desses sistemas. Como resultado, esboça-se a proposição de uma ferramenta para auxiliar os desenvolvedores na detecção dessas mensagens faltantes.*

1. Introdução

Sistemas de software estão em constante evolução através da adição de novas funcionalidades, correção de bugs e refatoração de código. Nesse processo, as APIs desses sistemas também estão sujeitas a mudanças. Quando essas APIs evoluem, seus clientes internos e externos também devem ser atualizados para se beneficiarem de interfaces melhores.

De modo a facilitar a atualização dos clientes, algumas boas práticas devem ser adotadas pelos desenvolvedores de APIs. Por exemplo, antes de serem removidas ou renomeadas, as APIs devem ser sempre depreciadas com mensagens claras para seus clientes de modo a manter sua compatibilidade. No entanto, na prática, existem evidências de que APIs são comumente depreciadas com mensagens obscuras ou mesmo

sem mensagens, dificultando a sua atualização por parte de clientes [Wu et al. 2010, Robbes et al. 2012, Hora et al. 2015].

Neste artigo, estuda-se um conjunto de questões relativas ao uso de mensagens de depreciação de APIs para melhor entender esse fenômeno. Tem-se como objetivo (i) mensurar a utilização de mensagens na depreciação de APIs e (ii) investigar a viabilidade da construção de uma ferramenta de recomendação dessas mensagens. Desse modo, estuda-se três questões de pesquisa centrais:

1. Com que frequência métodos são depreciados com mensagens de auxílio aos desenvolvedores? Com que frequência métodos são depreciados sem essas mensagens?
2. A quantidade de métodos depreciados sem mensagens tende a aumentar ou diminuir com passar do tempo?
3. Métodos depreciados com mensagens realmente sugerem como o desenvolvedor deve proceder?

Para responder essas questões analisa-se a evolução de 14 sistemas de software reais desenvolvidos em Java, tais como Hibernate, Apache Tomcat e Google Guava. Logo, as principais contribuições desse trabalho são: (1) apresenta-se uma caracterização da utilização de mensagens de depreciação de APIs em sistemas de software reais e (2) fornece-se uma base para recomendação dessas mensagens para desenvolvedores.

Esse artigo está organizado como descrito a seguir. A Seção 2 apresenta a metodologia proposta e a Seção 3 os resultados. Na Seção 4, são apresentadas as implicações desse estudo e na Seção 5 os riscos à validade. Finalmente, a Seção 6 discute trabalhos relacionados e a Seção 7 conclui o estudo.

2. Metodologia

Neste estudo foram analisados 14 sistemas *open-source* amplamente utilizados e escritos em Java. Para cada sistema foram analisadas três *releases*: inicial, intermediária e final (*i.e.,* última disponível). A Tabela 1 mostra a descrição dos sistemas, juntamente com as *releases* analisadas.

Em Java um método é depreciado utilizando-se a anotação *@Deprecated* e/ou a tag *@deprecated* inserida no Javadoc do respectivo método, conforme mostrado na Figura 1. Métodos depreciados através da tag *@deprecated* podem incluir mensagens em seu Javadoc sugerindo um novo método a ser utilizado através de tags *@link* e *@see*. Por outro lado, métodos depreciados por meio da anotação *@Deprecated* não possuem mensagens associadas. Nesse caso, um *warning* será emitido ao desenvolvedor pelo compilador, mas sem sugestão de como proceder.

Neste trabalho, utilizou-se a biblioteca JDT (*Java Development Tools*) para a detecção de métodos depreciados com e sem mensagens de auxílio aos desenvolvedores. Para isso, foi implementado um parser baseado em JDT para a coleta das ocorrências das anotações e das tags de depreciação em métodos, assim como suas respectivas mensagens Javadoc.

Table 1. Sistemas analisados

Sistema	Descrição	Release		
		Inicial	Inter.	Final
ElasticSearch	Servidor de buscas	1.3.7	1.4.3	1.5.2
Google Guava	Framework para manutenção de código	8.0	12.0	17.0
Netty	Framework para protocolos web	3.2.0	3.6.1	4.1.0
Apache Hadoop	Plataforma para sistemas distribuídos	1.0.4	2.4.1	2.7.0
Hibernate	Framework de persistência de dados	3.5.0	3.5.6	4.3.1
JUnit	Framework para testes	3.8.1	4.5	4.12
Apache Lucene	Biblioteca para busca textual	4.3.0	4.7.0	5.1.0
Spring Framework	Framework para aplicações web	1.0	3.0.0	4.1.0
Apache Tomcat	Servidor web	4.1.12	6.0.16	8.0.15
Log4J	Gerenciador de logs	1.0.4	1.2.13	2.2
Facebook Fresco	Gerenciador de imagens	0.1.0	0.3.0	0.5.0
Picasso	Gerenciador de imagens	1.0.0	2.0.0	2.1.1
Rhino	Implementação JavaScript em Java	1.4	1.6	1.7
Eclipse JDT	Infraestrutura para IDE do Eclipse	3.7	4.2.1	4.4.1

```java
/**
 * Does some thing in old style.
 *
 * @deprecated use {@link #new()} instead.
 */
@Deprecated
public void old() {
// ...
}
```

Figure 1. Exemplo de depreciação de método em Java

3. Resultados

3.1. Com que frequência métodos são depreciados com e sem mensagens de auxílio aos desenvolvedores?

Nesta questão de pesquisa estuda-se a frequência com que sistemas de software reais contém mensagens de depreciação de APIs. Dos 14 sistemas analisados, nove apresentam pelo menos um método depreciado sem mensagem. Esses sistemas são analisados com mais detalhes no restante do artigo.

A Tabela 2 mostra a quantidade de métodos depreciados com e sem mensagens na última *release* desses nove sistemas. Verifica-se, na prática, uma quantidade relevante de métodos depreciados sem mensagens. Por exemplo, no Apache Hadoop esse valor é de 42% dos métodos e no Netty 38%. Por outro lado, no Google Guava esse valor é de apenas 2%. Considerando todos os sistemas, dos 1,128 métodos depreciados, 278 (24%) não contém mensagens para auxiliar seus clientes.

Table 2. Número e percentagem de métodos depreciados com e sem mensagens.

Sistema	# Mét. Depreciados	# Com Mensagem	# Sem Mensagem
Elastic Search	77	55 (71%)	22 (29%)
Google Guava	168	165 (98%)	3 (2%)
Netty	80	50 (62%)	30 (38%)
Apache Hadoop	284	162 (57%)	122 (42%)
Hibernate	184	174 (95%)	10 (5%)
JUnit	29	23 (80%)	6 (20%)
Apache Lucene	79	61 (77%)	18 (23%)
Spring Framework	138	87 (63%)	51 (37%)
Apache Tomcat	89	73 (82%)	16 (18%)
Total	1128	850 (76%)	278 (24%)

3.2. A quantidade de métodos depreciados sem mensagens tende a aumentar ou diminuir com passar do tempo?

Nesta questão de pesquisa verifica-se se a quantidade de métodos depreciados sem mensagens aumenta ou diminui nas três *releases* analisadas (*i.e.,* inicial, intermediária e final).

A Tabela 3 mostra a quantidade desses métodos em cada uma das *releases*. A última coluna indica se o delta entre as *releases* foi acompanhado de um aumento (+) ou uma diminuição (–).

Table 3. Número e percentagem de métodos depreciados sem mensagens nas *releases* iniciais, intermediárias e finais.

Sistema	# Métodos Depreciados Sem Mensagens			Delta
	Inicial	Intermediária	Final	
Elastic Search	14 (36%)	30 (30%)	22 (29%)	– –
Google Guava	4 (13%)	7 (10%)	3 (2%)	– –
Netty	18 (46%)	72 (55%)	30 (38%)	+ –
Apache Hadoop	48 (30%)	102 (40%)	122 (42%)	+ +
Hibernate	0 (0%)	3 (4%)	10 (5%)	+ +
JUnit	0 (0%)	0 (0%)	6 (20%)	+
Apache Lucene	1 (3%)	12 (13%)	18 (23%)	+ +
Spring Framework	0 (0%)	6 (10%)	51 (37%)	+ +
Apache Tomcat	0 (0%)	4 (2%)	16 (18%)	+ +
Total	85 (14%)	236 (18%)	278 (24%)	+ +

Intuitivamente espera-se que a quantidade de métodos depreciados sem mensagens diminua, uma vez que as APIs desses sistemas se tornem mais robustas. No entanto, esse foi o caso de apenas três sistemas: ElasticSearch, Google Guava e Netty (entre a *release* intermediária e final). No ElasticSearch esse número caiu de 36% na *release* inicial para 29% na final. Percebe-se que a evolução do Google Guava inclui um esforço para utilização de mensagens: a quantidade de métodos sem mensagens de depreciação caiu de 13% para apenas 2%.

No entanto, a quantidade de métodos depreciados sem mensagens aumentou na maioria dos sistemas. Por exemplo, no Apache Hadoop, essa percentagem passou de 30%

na *release* inicial para 42% na final. No Hibernate, JUnit, Spring Framework e Apache Tomcat verifica-se que a percentagem inicial de 0% (*i.e.,* todos os métodos foram depreciados com mensagens). No entanto, na *release* final tem-se 5%, 20%, 37% e 18% de métodos depreciados sem mensagens, respectivamente. Considerando todos os sistemas, a percentagem subiu de 14% na verão inicial desses sistemas para 24% na final.

Uma possível explicação para a ausência de mensagens de depreciação é que os desenvolvedores podem estar adicionando mensagens na *release* imediatamente posterior. Por exemplo, eles podem em um primeiro momento depreciar o método (indicando que esse método será removido no futuro) e em um segundo momento (*i.e.,* na *release* seguinte) adicionar as mensagens. Desse modo, analisou-se as *releases inicial+1* e *intermediária+1* em busca de métodos que receberam mensagens. Especificamente, foi verificado a possibilidade de um método depreciado *somente* com a anotação @*Deprecated* receber posteriormente um Javadoc com a tag @*deprecated*. Como resultado, constatou-se que nenhum dos métodos depreciados sem mensagem foi refatorado na *release* seguinte para receber tais mensagens.

3.3. Métodos depreciados com mensagens realmente sugerem como o desenvolvedor deve proceder?

Nesta questão de pesquisa investiga-se se métodos depreciados com mensagens incluem tags (@*see* ou @*link*) ou palavras chaves (*use*) que apontem para sua substituição.

A Tabela 4 resume esses dados para a última *release* de cada sistema. No JUnit 91% dos métodos depreciados com mensagens incluem elementos para ajudar os desenvolvedores. No Google Guava, embora exista um esforço para incluir mensagens (conforme mostrado na Questão 2), verifica-se que apenas 38% das mensagens incluem esses elementos. Já no Apache Tomcat, as tags padrão @*see* e @*link* nunca são utilizadas, mas sim a palavra chave *use* em 29% das mensagens. Em suma, verifica-se uma quantidade relevante de métodos depreciados com os elementos considerados (*e.g.,* 54% utilizam a tag @*link*), no entanto, uma grande parte dessas mensagens ainda são incompletas.

Table 4. Número e percentagem de métodos depreciados com mensagens utilizando as tags *@see* e *@link* e a palavra *use*.

Sistema	# Métodos Depreciados Com Mensagens		
	@see	@link	use
Elastic Search	0 (0%)	36 (65%)	36 (65%)
Google Guava	1 (1%)	62 (38%)	62 (38%)
Netty	0 (0%)	40 (50%)	40 (50%)
Apache Hadoop	11 (7%)	136 (84%)	136 (84%)
Hibernate	4 (2%)	113 (65%)	113 (65%)
JUnit	0 (0%)	21 (91%)	21 (91%)
Apache Lucene	2 (3%)	36 (59%)	36 (59%)
Spring Framework	36 (41%)	36 (41%)	23 (26%)
Apache Tomcat	0 (0%)	0 (0%)	21 (29%)
Total	54 (6%)	480 (54%)	488 (55%)

As Figuras 2 e 3 contém trechos de código exemplificando a depreciação de métodos no sistema Spring Framework. A Figura 2 mostra a utilização da palavra chave

use acompanhado da tag @*link* indicando o método que irá substituir o depreciado. Já na Figura 3 verifica-se o uso das tags @*see* e @*link*. Nota-se que pode haver sobreposição entre as duas formas de depreciar métodos, o que justifica o fato de a soma dos totais de métodos que utilizam as tags @*see* e @*link* e a palavra chave *use* ultrapassar 100%.

```
/**
 * Merge the specified Velocity template with the given model and write
 * the result to the given Writer.
 * @param velocityEngine VelocityEngine to work with
 * @param templateLocation the location of template, relative to Velocity's resource loader path
 * @param model the Map that contains model names as keys and model objects as values
 * @param writer the Writer to write the result to
 * @throws VelocityException if the template wasn't found or rendering failed
 * @deprecated Use {@link #mergeTemplate(VelocityEngine, String, String, Map, Writer)}
 * instead, following Velocity 1.6's corresponding deprecation in its own API.
 */
@Deprecated
public static void mergeTemplate(
                VelocityEngine velocityEngine, String templateLocation, Map<String, Object> model, Writer writer)
                throws VelocityException {

        VelocityContext velocityContext = new VelocityContext(model);
        velocityEngine.mergeTemplate(templateLocation, velocityContext, writer);
}
```

Figure 2. Depreciação utilizando *use* e *@link*

```
/**
 * Determine whether the given attribute is eligible for being
 * turned into a corresponding bean property value.
 * <p>The default implementation considers any attribute as eligible,
 * except for the "id" attribute and namespace declaration attributes.
 * @param attribute the XML attribute to check
 * @see #isEligibleAttribute(String)
 * @deprecated in favour of {@link #isEligibleAttribute(org.w3c.dom.Attr, ParserContext)}
 */
@Deprecated
protected boolean isEligibleAttribute(Attr attribute) {
        return false;
}
```

Figure 3. Depreciação utilizando *@see* e *@link*

4. Implicações

Nesta seção são apresentadas duas implicações do presente estudo. A Questão 1 confirma que APIs são frequentemente depreciadas sem mensagens de substituição para ajudar os clientes. A Questão 2 mostra que esse problema se agrava com o passar do tempo: na maioria dos sistemas analisados a quantidade de APIs sem mensagens aumenta. Isso pode ser explicado pela complexidade e tamanho dos sistemas analisados. De fato, trabalhar nessas APIs pode não ser uma tarefa simples, mas sim envolver diversos desenvolvedores com os mais diversos níveis de conhecimento, sendo difícil manter consistência durante sua evolução [Wu et al. 2010, Robbes et al. 2012, Hora et al. 2015]. Nesse sentido, existe um esforço da literatura em minerar evolução de APIs (*e.g.,* [Kim and Notkin 2009, Hora et al. 2013, Meng et al. 2013, Hora et al. 2014], mas não no contexto de ajudar na depreciação de métodos. Assim, apresenta-se a seguinte implicação:

Implicação 1: Uma ferramenta de recomendação de mensagens de depreciação pode ser construída para auxiliar tanto os desenvolvedores das APIs quanto seus clientes. Essas mensagens podem ser inferidas através da mineração das reações dos clientes, *i.e.,* pode-se aprender a solução adotada pelos clientes na ausência de mensagens de substituição de métodos.

A Questão 3 mostra que uma quantidade relevante de métodos são depreciados com mensagens utilizando certas tags da linguagem Java. Quando a depreciação é realizada com uma mensagem bem estruturada, o método depreciado pode ser diretamente substituído pelo recomendado na mensagem. Essa informação fornece a base para seguinte implicação:

> *Implicação 2*: A qualidade de uma ferramenta de recomendação de mensagens de depreciação pode ser verificada através de sua correção em identificar mensagens de métodos depreciados *com* mensagens. Em outras palavras, os métodos depreciados com mensagens podem ser utilizados como um oráculo para mensurar a acurácia da ferramenta em identificar mensagens válidas.

5. Riscos à Validade

Validade Externa. Os resultados desse estudo estão restritos a análise de sistemas Java e não podem ser generalizados para outras linguagens. Adicionalmente, como os sistemas analisados são *open-source*, pode ocorrer resultados diferentes em sistemas comerciais. Além disso, o estudo descrito no presente trabalho envolveu 14 sistemas. A princípio essa quantidade poderia comprometer a generalização das conclusões obtidas. Porém, esse risco é atenuado pelo fato de os sistemas analisados serem relevantes e de alta complexidade.

Validade Interna. Neste estudo coletou-se mensagens relativas a depreciação dos métodos das APIs analisadas. Para isso foi implementado um *parser* baseado na biblioteca JDT, que é desenvolvido pelo Eclipse para lidar com AST. Logo, o fato do parser implementado ser baseado no Eclipse JDT o deixa com menos chances de erros.

Validade de Construção. Foram analisadas para cada sistema três *releases*: inicial, intermediária e final. Essas três *releases* não caracterizam todo o desenvolvimento desses sistemas. No entanto, elas caracterizam uma visão geral de suas evoluções.

6. Trabalhos Relacionados

Existem alguns trabalhos que mostram evidências de que APIs são comumente depreciadas com mensagens obscuras ou mesmo sem mensagens, dificultando o processo de atualização dos clientes [Robbes et al. 2012, Hora et al. 2015]. Esses trabalhos analisam um ecossistema com sistemas implementados em Smalltalk e concluem que a qualidade das mensagens deve ser melhorada. No presente estudo, essa análise foi estendida para sistemas Java (Questão 1). Além disso, o presente estudo aprofundou outras questões não abordadas nos trabalhos relacionados: foi investigado a evolução dos métodos depreciados sem mensagens (Questão 2) e se as mensagens realmente sugerem como os desenvolvedores devem proceder (Questão 3).

A literatura propõe diversas abordagens para lidar com evolução de APIs. Por exemplo, essa tarefa pode ser realizada com a ajuda IDEs modificadas [Henkel and Diwan 2005], a ajuda dos desenvolvedores das APIs [Chow and Notkin 1996] ou através da mineração de repositórios de software [Xing and Stroulia 2007, Kim and Notkin 2009, Wu et al. 2010, Hora et al. 2012, Hora et al. 2013, Meng et al. 2013, Hora et al. 2014, Hora et al. 2015, Hora and Valente 2015]. No entanto, nenhum desses trabalhos aborda diretamente o problema da ausência de mensagens na depreciação.

7. Conclusões

Neste trabalho caracterizou-se a utilização de mensagens de depreciação de APIs por meio da análise de 14 sistemas reais implementados em Java. Verificou-se que existe uma parcela relevante (24%) de métodos depreciados sem essas mensagens. Também foi constatado que esse problema se agrava ao longo da evolução dos sistemas analisados: na maioria dos sistemas a quantidade de métodos sem mensagens aumenta (de modo geral de 14% para 24%). A partir desses resultados verificou-se a viabilidade da construção de uma ferramenta para recomendação de mensagens de depreciação para auxiliar os desenvolvedores nas suas atividades de migração de APIs.

Agradecimentos: Esta pesquisa é financiada pela FAPEMIG e pelo CNPq.

Referências

Chow, K. and Notkin, D. (1996). Semi-automatic update of applications in response to library changes. In *International Conference on Software Maintenance*.

Henkel, J. and Diwan, A. (2005). Catchup!: Capturing and replaying refactorings to support API evolution. In *International Conference on Software Engineering*.

Hora, A., Anquetil, N., Ducasse, S., and Allier, S. (2012). Domain Specific Warnings: Are They Any Better? In *International Conference on Software Maintenance*.

Hora, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2013). Mining System Specific Rules from Change Patterns. In *Working Conference on Reverse Engineering*.

Hora, A., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2014). APIEvolutionMiner: Keeping API Evolution under Control. In *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*.

Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). How do developers react to API evolution? the Pharo ecosystem case. In *International Conference on Software Maintenance and Evolution*.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of API popularity and migration. In *International Conference on Software Maintenance and Evolution*. `http://apiwave.com`.

Kim, M. and Notkin, D. (2009). Discovering and Representing Systematic Code Changes. In *International Conference on Software Engineering*.

Meng, N., Kim, M., and McKinley, K. S. (2013). Lase: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering*.

Robbes, R., Lungu, M., and Röthlisberger, D. (2012). How do developers react to API deprecation? The case of a smalltalk ecosystem. In *International Symposium on the Foundations of Software Engineering*.

Wu, W., Gueheneuc, Y.-G., Antoniol, G., and Kim, M. (2010). Aura: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering*.

Xing, Z. and Stroulia, E. (2007). Api-evolution support with diff-catchup. *Transactions on Software Engineering*, 33(12).

Experimental Evaluation of Code Smell Detection Tools

Thanis Paiva[1], Amanda Damasceno[1], Juliana Padilha[1], Eduardo Figueiredo[1], Claudio Sant'Anna[2]

Computer Science Department – Federal University of Minas Gerais (UFMG)

Computer Science Department – Federal University of Bahia (UFBA)

{thpaiva, amandads, juliana.padilha, figueiredo}@dcc.ufmg.br, santanna@dcc.ufba.br

Abstract. *Code smells are code fragments that can hinder the evolution and maintenance of software systems. Their detection is a challenge for developers and their informal definition leads to the implementation of multiple detection techniques by tools. This paper aims to evaluate and compare three code smell detection tools, namely inFusion, JDeodorant and PMD. These tools were applied to different versions of a same system. The results were analyzed to answer four research questions related to the evolution of the smells, the relevance of the smells detected, and the agreement among tools. This study allowed us to understand the evolution of code smells in the target system and to evaluate the accuracy of each tool in the detection of three code smells: God Class, God Method, and Feature Envy.*

1. Introduction

Code smells are symptoms that something may be wrong in the system code [Fowler 1999]. They can degrade system quality aspects, such as maintainability, and potentially introduce flaws [Yamashita and Counsell 2013]. Different studies report that the cost of software maintenance is about 70-80% of the total project cost [Dehagani and Hajrahimi 2013]. Therefore, it is important to detect code smells throughout the software development. However, code smell detection is a complex, tedious and error prone task. In this context, tools for automatic detection of code smells can assist developers in the identification of affected entities, facilitating the detection task. This goal is accomplished by the implementation of different detection techniques that allow the tools to highlight the entities most likely to present code smells.

Nowadays, there are an increasing number of software analysis tools available for detecting bad programming practices [Fontana et al. 2012] [Murphy-Hill and Black 2010] [Tsantalis et al. 2008] and, in general, there is increasing awareness of software engineers about the structural quality of features under development. The question is how to assess and compare tools to select which is more efficient in a given situation. However, evaluation of the effectiveness of tools for detecting code smells presents some challenges [Fontana et al. 2012]. First, there are different interpretations for each code smell, given the ambiguity and sometimes vagueness of their definitions. This lack of formalism prevents different tools to implement the same detection technique for a specific code smell. Second, these techniques generate different results, since they are usually based on the computation of a particular set of combined metrics, ranging from standard object-oriented metrics to metrics defined in ad hoc way for the smell detection purpose [Lanza and Marinescu 2006]. Finally, even if the same metrics are used, the threshold values might be different because they are established considering different factors, such as system domain and its size, organizational practices and the experience

and understanding of software engineers and programmers that define them. Changing the threshold has a large impact on the number of code smells detected.

The previous factors make it difficult to validate the results generated by different techniques, which is made only for small systems and few code smells [Mäntylä 2005] [Moha et al. 2010] [Murphy-Hill and Black 2010]. For instance, Fontana [2012] investigated six code smells in one software system, named GanttProject. Their study showed that a more effective analysis requires a greater understanding of the system as well as its code smells. The difficulty lies not only in the different interpretations of code smells, but also in the manual identification of the code smells, that is also a challenge. Therefore, it is difficult to find open-source systems with validated lists of code smells to allow further analysis.

Using an approach similar to Fontana [2012], this paper presents an in depth analysis of the identification and evolution of three code smells in different versions of a software system, named MobileMedia [Figueiredo et al. 2008]. We compare the accuracy in terms of recall and precision of three code smell detection tools, namely inFusion[1], JDeodorant[2] [Tsantalis et al. 2008], and PMD[3]. This paper also presents a measurement of agreement among the analyzed tools. We focus our investigation on three code smells detected by these tools: God Class [Riel 1996], God Method [Fowler 1999], and Feature Envy [Fowler 1999]. We found that in spite of a high agreement among the tools, their individual accuracy varies depending on the code smell being analyzed.

The rest of this paper is organized as follows. Section 2 introduces the code smells and the detection tools evaluated. Section 3 describes the study settings. Section 4 presents the results and tries to answer research questions based on statistical analysis. Section 5 discusses the main threats to the validity of this study and the related work while Section 6 concludes this paper and points out directions for future work.

2. Code Smell Definitions and Detection Tools

Smell Definitions. Code smells were proposed by Kent Beck in Fowler's book [Fowler 1999] as a mean to diagnose symptoms that may be indicative of something wrong in the system code. In this paper, we focus on three code smells. *God Class* describes an object that knows too much or does too much [Riel 1996]. *God Method* represents a method that has grown too much [Fowler 1999] and tends to centralize the functionality of a class. *Feature Envy* represents a method that seems more interested in a class other than the one it actually is in [Fowler 1999]. These smells were selected because they are the three predominant smells in the target system [Padilha et al. 2014] that can be detected by at least two of the following evaluated tools.

Detection Tools. In this paper we evaluate three code smell detection tools: inFusion[1], JDeodorant[2], and PMD[3]. They were selected because they are available for download and actively developed and maintained. Their results are easily accessible and they are able to detect the three predominant code smells in our target system: MobileMedia. More information about the tools evaluated is available on the project website[4].

[1] Available at http://www.intooitus.com/products/infusion
[2] Available at http://jdeodorant.com/
[3] Available at http://pmd.sourceforge.net/
[4] Available at http://homepages.dcc.ufmg.br/~thpaiva/vem2015

inFusion[1]. This is a commercial standalone tool for Java, C, and C++ that detects 22 code smells, including the three smells of our interest: God Class, God Method, and Feature Envy. The detection techniques for all the smells were initially based in the detection strategies defined by Lanza and Marinescu [2006], and then successively refined using real code.

JDeodorant[2] [Tsantalis et al. 2008]. This is an open-source Eclipse plugin for Java that detects four code smells: God Class, God Method, Feature Envy, and Switch Statement. The detection techniques are based in refactoring opportunities, for God Class and Feature Envy and slicing techniques, for God Method [Fontana et al. 2012].

PMD[3]. This is an open-source tool for Java and an Eclipse plugin that detects many problems in Java code, including two of the smells of our interest: God Class and God Method. The detection techniques are based on metrics. For God Class, it relies on the detection strategy defined by Lanza and Marinescu [2006] and, for God Method, a single metric is used: NLOC.

3. Study Settings

Target System. Our study involved the MobileMedia system [Figueiredo et al. 2008], a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices. This system was selected because it has been previously used in other maintainability-related studies [Figueiredo et al. 2008] [Macia et al. 2012], and we have access to their developers and experts. Therefore, we were able to recover its reference list of actual code smells. Our study involved nine versions object-oriented (1 to 9) of MobileMedia, ranging from 1 KLOC to a little over 3 KLOC. Table 1 shows for each version: the number of classes, methods, and lines of code. It also contains the total number of God Classes (GC), God Methods (GM), and Feature Envy (FE) found in each system version.

Table 1. MobileMedia System Information

	Size Metrics of MobileMedia			Code Smells in the Reference List			
V	# of Classes	# of Methods	LOC	GC	GM	FE	Total
1	24	124	1159	3	9	2	14
2	25	143	1316	3	7	2	12
3	25	143	1364	3	6	2	11
4	30	161	1559	4	8	2	14
5	37	202	2056	5	8	2	15
6	46	238	2511	6	9	2	17
7	50	269	3015	7	7	2	16
8	50	273	3167	9	7	2	18
9	55	290	3216	7	6	3	16

Reference List Protocol. Two experts used their own strategy for detecting, individually and manually, the code smells in the system's classes and methods. As a result, two lists of entities were created. These lists were merged and the findings discussed with a developer of the system to achieve a consensus and validate the entities that present a code smell. The result of this discussion generated the final reference list used in this paper.

Research Questions. In this study, first we explore the presence of code smells in the system. For that purpose, we formulate the first two research questions (RQ1 and RQ2) presented below. We then analyze and compare the three detection tools based on two additional research questions (RQ3 and RQ4).

RQ1. *The number of code smells increase over time?*

RQ2. *How the code smells evolve with the system evolution?*

RQ3. *What is the accuracy of each tool in identifying relevant code smells?*

RQ4. *Do different detection tools agree for the same code smell when applied to the same software system?*

4. Statistical Analysis of Results and Discussions

4.1. Summary of Detected Code Smells

Table 2 shows the total number of code smell instances identified in the nine versions of MobileMedia by each tool. The most conservative tool is inFusion, with a total of 28 code smell instances. PMD is less conservative, detecting a total of 24 instances for God Class and God Method, in contrast with the 20 detected by inFusion. JDeodorant is more aggressive in its detections strategy by reporting 254 instances. That is, it detects more than nine times the amount of smells of the most conservative tools, inFusion and PMD. The reference list contains a total of 133 instances distributed among the nine versions of MobileMedia. The complete raw data is available on the project website[4].

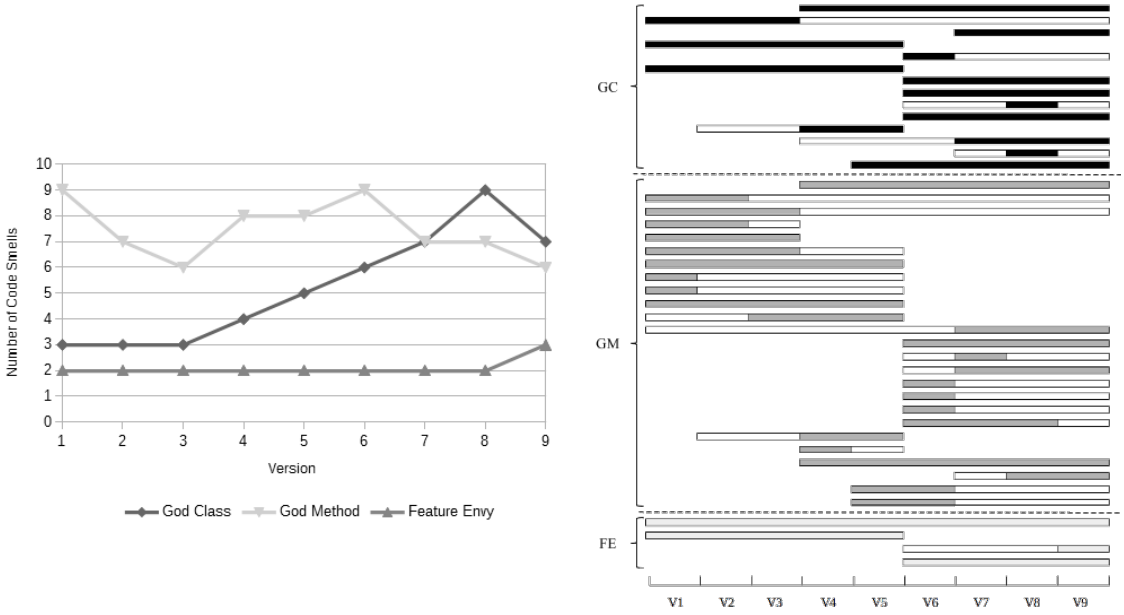
Table 2. Total number of identified code smells among versions by detection tools

Code Smell	inFusion	JDeodorant	PMD	Reference List
God Class	3	85	8	47
God Method	17	100	16	67
Feature Envy	8	69	-	19
Total	**28**	**254**	**24**	**133**

4.2. Evolution of Code Smells

This section aims to answer the research questions RQ1 and RQ2 using the code smell reference list. Focusing on RQ1, the results summarized in Figure 1a confirmed that only the number of God Class increases with the system's evolution. That was expected, since the evolution of the system includes new functionalities and God Classes tend to centralize them. The number of God Methods varies, but the last version has fewer instances than the first one. This variation is due to more frequent modification of methods between versions. For Feature Envy, the number of instances remained constant from version 1 up to 8. Only the final version has one additional instance.

To answer RQ2, we investigate the evolution of each code smell in Figure 1b. In Figure 1b each rectangle in a row represents the lifetime of an entity (class or method) in the system. The shaded area indicates the presence of a code smell. Each column consists of a particular system version. A rectangle is omitted if the entity does not exist in that particular version. We found that for 10 out of 14 God Class instances, the problem originates with the class. This result is aligned with recent findings [Tufano et al. 2015]. For the other 4 instances, the class becomes a code smell in later versions. For God Method, there is a lot of variation among versions, some methods are created with a code smell (19 of 25) and others become a code smell with the evolution of the system (6 of 25). However, when a smell is introduced in a method, it tends to be removed permanently in subsequent versions. For Feature Envy, in 3 out of 4 instances, the smell originated with the method and persisted during its entire existence. Only one method was created without Feature Envy and evolved to present that code smell.



(a) Number of code smells per version. **(b) Evolution of code smells**

Figure 1. Code Smells in MobileMedia

4.3. Analysis of Precision and Recall

This section aims to answer RQ3 presented in Section 3. To assess the tools accuracy to detect relevant code smell, we calculated the precision and recall based on the code smell reference list. A relevant code smell is a smell from the code smell reference list. Table 3 shows the average of recall and precision considering all versions for each tool and code smell analyzed. In general, for all code smells, JDeodorant has the highest average recall, followed by PMD and inFusion. In spite of its high recall, JDeodorant is also the tool that reports the highest number of false positives. That is, it has a lower precision. Consequently, the results require a greater validation effort by the developer.

For God Class and God Method, PMD and inFusion have a similar accuracy, i.e., lower recall, but higher precisions when compared to JDeodorant. They achieve 100% precision for God Method. Higher precision reduces greatly the validation effort of the programmer, but at the risk of missing relevant code smell.

For Feature Envy, inFusion had the worst overall accuracy with 0% of recall and precision. JDeodorant had a better accuracy for Feature Envy when compared to inFusion, although these were the worst values for JDeodorant. This can be an indicator that Feature Envy is a more complex code smell to be automatically detected, when compared to seemingly less complex smells, such as God Class and God Method.

Table 3. Average Recall and Precision for inFusion, JDeodorant and PMD

	Recall			Precision		
Code Smell	**inFusion**	**JDeodorant**	**PMD**	**inFusion**	**JDeodorant**	**PMD**
God Class	9%	58%	17%	33%	28%	78%
God Method	26%	50%	26%	100%	35%	100%
Feature Envy	0%	48%	-	0%	13%	-

4.4. Analysis of Agreement

This section aims to answer RQ4. We calculated the agreement among tools to investigate whether different detection tools return the same results for different detection algorithms. God Class and God Method are detected by inFusion, JDeodorant, and PMD. Feature Envy is detected only by inFusion and JDeodorant.

Table 4 shows the calculated overall agreement (OA) of the tools and the AC_1 *statistic* [Gwet 2014], which adjusts the overall agreement probability for chance agreement with a 95% confidence interval (CI). The overall agreement (OA) among tools is high, ranging from 78.38 % to 99.51%. However, the high agreement is due to the greater agreement on true negatives, i.e., most entities are categorized as not containing any smell. Comparing the tools results with the code smell reference list, we concluded that the occurrence of a smell in the system is indeed rare, as indicated by the detection tools.

Table 4. Overall Agreement (OA) and AC1 Statistics of the Analyzed Tools

	God Class			God Method			Feature Envy		
V	OA	AC1	95% CI	OA	AC1	95% CI	OA	AC1	95% CI
1	80.56%	0,764	[0.576,0.953]	94.62%	0,943	[0.907,0.979]	91.13%	0,903	[0.843,0.963]
2	81.33%	0,757	[0.565,0.949]	94.80%	0,946	[0.914,0.979]	92.20%	0,915	[0.862,0.967]
3	81.33%	0,757	[0.565,0.949]	94.91%	0,946	[0.914,0.979]	92.36%	0,916	[0.865,0.968]
4	82.22%	0,788	[0.631,0.945]	96.71%	0,966	[0.941,0.990]	93.83%	0,934	[0.891,0.976]
5	78.38%	0,732	[0.574,0.890]	95.40%	0,951	[0.925,0.978]	99.51%	0,995	[0.985,1.000]
6	85.51%	0,833	[0.724,0.942]	97.77%	0,977	[0.961,0.993]	97.49%	0,974	[0.953,0.995]
7	86.67%	0,848	[0.749,0.947]	96.79%	0,966	[0.948,0.985]	97.04%	0,970	[0.948,0.991]
8	82.67%	0,791	[0.672,0.911]	95.62%	0,954	[0.932,0.975]	98.18%	0,981	[0.965,0.998]
9	83.03%	0,797	[0.685,0.909]	97.04%	0,969	[0.952,0.986]	97.95%	0,979	[0.962,0.996]

The AC1 statistic is a robust agreement coefficient alternative to *Kappa* [Gwet 2014] and other common statistics for inter-rater agreement. It takes a value between 0 and 1, and communicates levels of agreement using the Altman's benchmark scale for Kappa [McCray 2013], that classifies agreement levels into Poor (< 0.20), Fair (0.21 to 0.40), Moderate (0.41 to 0.60), Good (0.61 to 0.80), and Very Good (0.81 to 1.00) [Altman 1991]. We found an AC1 “Very Good” for all smells and versions, with the exception of God Class in versions 6 and 7 with an AC1 “Good” with high precision, i.e., narrow confidence intervals.

The variation in the results is due to the implementation of different detection techniques or the variation in threshold values for the same technique. In our case, each tool uses a different detection technique, so we did not analyze the impact of different thresholds in the results. However, we are aware that they influence the results.

5. Threats to Validity and Related Work

Threats to Validity. Some limitations are typical of studies like ours, so we discuss the study validity with respect to common threats to validity. *Internal Validity* refers to threats of conclusions about the cause and effects in the study [Wohlin et al. 1999]. The main factors that could negatively affect the internal validity of the experiment are the size of the subject programs and possible errors in the transcription of the result of tool analysis. The subjects of our analysis are the nine versions of the MobileMedia that are small size programs. About transcription errors, the tools analyzed generate outputs in different formats. To avoid transcriptions errors we have reviewed all the data multiple

times. *External Validity* concerns the ability to generalize the results to other environments [Wohlin et al. 1999]. MobileMedia is a small open source system developed by a small team with an academic focus, therefore is not extendable to real large scale projects. This limits the generalization of our results. *Conclusion validity*, on the other hand, concerns the relation between the treatments and the outcome of the experiment [Wohlin et al. 1999]. This involves the correct analysis of the results of the experiment, measurement reliability and reliability of the implementation of the treatments. To minimize this threat, we discussed the results data to make a more reliable conclusion.

Related Work. Fontana [2012] presented an experimental evaluation of multiple tools, code smells and programs. Similarly, we evaluated tool agreement. However, we used the AC1 statistic, a more robust measure than the Kappa Coefficient. We also analyzed each tools precision and recall, unlike Fontana [2012]. Chatzigeorgiou and Manakos [2010] focused their analysis in evolution of code smells. In our work we also conduct this analysis, but at a higher level and not so focused in maintenance activities and refactoring.

6. Conclusions and Future Work

Comparing tools is a difficult task, especially when involves informal concepts, such as code smells. The different interpretations of code smell by researchers and developers leads to tools with distinct detection techniques, influencing the results and, consequently, the amount of time spent with validation.

In this paper we analyzed the code smells in the nine versions of MobileMedia to evaluate the accuracy of three code smell detection tools, namely inFusion, JDeodorant, and PMD to detect three code smells, God Class, God Method and Feature Envy. First we analyzed the evolution of MobileMedia code smells and found that the number of God Class increases, while the number of methods varies between versions and the number of Feature Envy remains constant. We also found that when a smell is when a smell is introduced in a method it tends to be removed permanently in subsequent versions.

Using MobileMedia as target system, we evaluated the tools accuracy and realized that there is a trade-off between the number of correctly identified entities and time spent with validation. For all smells, JDeodorant identified most of the correct entities, but generated reports with more false positives. This increases the validation effort, but captures the majority of the affected entities. On the other hand, PMD and inFusion identified more correct entities, however, also generated reports without some affected entities. This reduces validation effort, but neglects to highlight potential code smells. In future work, we would like to investigate more the evolution of other code smells in a system and how their evolution is related to maintenance activities.

Acknowledgments. This work was partially supported by CAPES, CNPq (grant 485907/2013-5), and FAPEMIG (grant PPM-00382-14).

References

Altman, D. G. (1991). “Practical Statistics for Medical Research”, Chapman & Hall, London.

Chatzigeorgiou, A. and Manakos, A. (2010). “Investigating the Evolution of Bad Smells in Object-Oriented Code”, In: Seventh Intl Conf. Quality of Information and Communications Technology.

Dehagani, S.M.H. and Hajrahimi, N. (2013). “Which Factors Affect Software Projects Maintenance Cost More?”, Acta Informatica Medica, vol.21, no.1, p.63.

Figueiredo, E. et al. (2008). “Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability”, Proc. of ICSE, p.261-270.

Fontana, F. A. et al. (2012). “Automatic Detection of Bad Smells in Code: An Experimental Assessment.”, Journal of Object Technology 11(2): 5: p.1-38

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

Gwet, K (2014). “Handbook of Inter-Rater Reliability: The Definite guide to Measuring the Extent of Agreement Among Raters”, Advanced Analytics, USA.

Lanza, M. and Marinescu, R. (2006). “Object-Oriented Metrics in Practice”. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Macia, I. et al.(2012) “Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?”, Proc. of AOSD, p.167-178

Mäntylä, M.V. (2005). “An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Inter-rater Agreement.”, In: Proc. Intl Symposium on Empirical Software Engineering, p. 287–296.

McCray, G. (2013). “Assessing Inter-Rater Agreement for Nominal Judgement Variables”, Paper presented at the Language Testing Forum. Nottingham, p.15-17.

Moha, N. et al. (2010). “From A Domain Analysis to the Specification and Detection of Code and Design Smells”, Formal Aspects of Computing, 22: p.345–361.

Murphy-Hill, E. and Black, A. (2010). “An Interactive Ambient Visualization for Code Smells”, In: Proceedings of the 5th international symposium on Software visualization, (SoftVis), p.5–14.

Padilha, J. et al. (2014). “On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study”. In proc. of the International Conference on Advanced Information Systems Engineering (CAiSE). Thessaloniki, Greece.

Riel, A.J. (1996). “Object-Oriented Design Heuristics”, Addison-Wesley Profess.

Tsantalis, N.; Chaikalis, T.; Chatzigeorgiou, A. (2008). “JDeodorant: Identification and Removal of Type-Checking Bad Smells”. In proc. of European Conference on Software Maintenance and Reengineering (CSMR).

Tufano, M. et al. (2015). “When and Why Your Code Starts to Smell Bad”. In proc. of International Conference on Software Engineering (ICSE). Firenze, Italy.

Wohlin, C. et al. (1999). "Can the Personal Software Process be used for Empirical Studies?", In: Proceedings ICSE workshop on Empirical Studies of Software Development and Evolution, Los Angeles, USA.

Yamashita, A. and Counsell, S. (2013). “Code Smells as System-Level Indicators of Maintainability: an Empirical Study”, Journal Systems and Software, p.2639-2653.

On Mapping Goals and Visualizations: Towards Identifying and Addressing Information Needs

Marcelo Schots, Cláudia Werner

Systems Engineering and Computing Program (PESC)
Federal University of Rio de Janeiro (COPPE/UFRJ) – Rio de Janeiro, RJ – Brazil

{schots,werner}@cos.ufrj.br

Abstract. *The design of visual metaphors and the development of visualization tools have a key difference: while the former is a creative process to produce novel and generic approaches, the latter is a goal-oriented process. Ensuring that a tool properly maps the data required to achieve its goals into visual attributes is not trivial, since there are several abstraction gaps to be addressed. The mapping structure presented in this paper aims to provide developers of visualization tools with a focused and cautious decision making. Its use in the design of Zooming Browser, a tool for performing software reuse analytics, enabled to check if the tool could help answering the established questions before evaluating it with the intended audience (i.e., stakeholders).*

1. Introduction and Motivation

The design of visualizations is a process that usually requires insights and/or creativity, and can occur in several ways. Information visualization designers may create concepts from the real world to represent abstract entities, or (more usually) they may try to combine knowledge on existing paradigms, strategies, and techniques to produce novel approaches. The produced visual metaphors are usually general, enabling their use for several purposes (as shown in [Abuthawabeh et al. 2013]) in diverse fields of interest.

On the other hand, the development of visualization tools goes the opposite way. Developers[1] recall existing abstractions trying to find out how to (better) depict the available/necessary data based on their characteristics. Furthermore, there is a purpose in mind when creating visualization tools, *i.e.*, there are specific *goals* to be met. Thus, not only there is the need to represent data using proper abstractions; it must be made in such a way that it can help somebody (audience) to do something (tasks). If this premise is neglected, the tool will be either useless or not fully meet the needs for which it is created, leading users to resort to other sources of information or additional tools.

It is important to ensure, among other aspects, that the visualization tool under development fits the established needs, properly mapping the data required to achieve the goals into corresponding visual attributes. However, this is not trivial, since there are several abstraction gaps to be addressed. The mapping of goals and visualizations cannot be made instantly, as this brings a considerable risk of overlooking important intermediary decisions; it must be decomposed into stages and performed carefully.

[1] In this paper, *developers* include roles who take decisions in the *development* of visualization tools (*e.g.*, requirements engineer, designer, programmer etc.), ranging from the tool goals to the visualizations to use.

This necessity became more evident during the design of Zooming Browser, a visualization tool for performing software reuse analytics [Schots 2014]. Its goals were based on findings from our previous studies, which pointed out limitations on existing visualization approaches for supporting software reuse [Schots & Werner 2014a] and needs identified in the software industry [Schots & Werner 2014b]. We wanted to assure that Zooming Browser visualizations would be actually helpful in answering some reuse-related questions, accomplishing the established goals. To this end, we realized that we needed to recognize which tasks users should perform to answer these questions, which data would be necessary, and how to map the data into the vast visualization space. This resulted in a mapping structure created to guide this process.

In this paper, we present a staged process for mapping *user* (or *organization*) *goals* to the *visualizations* that can help achieving such goals. The purpose of this mapping is not to enforce a set of guidelines on how to perform each stage, nor to point out what would be the best visualization for some goal/task/data. Instead, we want to encourage developers to perform a more focused and cautious decision making (not tied to any particular methodology) on each mapping stage, towards an anticipated assessment of the usefulness of their visualization tools before evaluating them with the intended audience. For illustrating how each stage can be carried out, we present some examples of the mapping performed during the development of Zooming Browser.

2. Related Work

Before elaborating the mapping structure, we analyzed related work to find out whether existing solutions would meet our needs. Some of these works are presented as follows.

The well-known and largely applied Goal-Question-Metric (GQM) approach [Basili et al. 1994] comprises the setting of goals, the derivation of questions from such goals, and the choice of metrics to answer these questions. GQM was the first attempt to derive the necessary data for Zooming Browser. However, we soon recognized that a metric or a set of metrics is usually not enough to answer all the questions developers have. It becomes necessary to perform some kind of task to find out the answers to those questions. Besides, GQM does not aim at mapping metrics and visualizations.

Some approaches offer a customizable mapping between visual elements and data. CogZ [Falconer et al. 2009], for instance, is a set of tools that provides a module for the rapid development of specific visualizations for ontologies. It provides drag and drop mechanisms for mapping concepts (ontology terms) to visual representations. Apart from the benefits and the customization facilities, the mapping process starts from the data, not focusing on the goals that led to choosing such data.

Beck et al. (2013) aim at helping visualization experts to choose visualization techniques for dynamic graph visualization scenarios. Profiles reflecting different aesthetic criteria describe both the techniques and the application: their similarity shows how appropriate a visualization technique is for such application (it may be necessary to refine profiles or consider other criteria to achieve a best match). A solid knowledge of visualization techniques and significant experience in visualization design are required.

These approaches support particular stages of the mapping process, but none of them provide the full picture on the mapping between a set of goals and visualizations.

3. The Zooming Browser Tool

Nowadays, developers write less code and consume more reusable code. Software reuse has become present in the daily routine of software developers, yet mostly in an ad-hoc or pragmatic way. However, it is central to consider the importance of *reuse awareness*, *i.e.*, knowing what is going on in the reuse scenario. It helps deciding whether a given asset should be reused, or communicating problems identified in any kind of reusable asset to its producers and consumers, among other benefits. However, achieving reuse awareness is challenging, especially because of the lack of tool support to this purpose.

The Zooming Browser tool aims at providing reuse awareness to support reuse managers[2] and developers in performing reuse-related tasks, both in the context of a software project (*e.g.*, the decision to incorporate or upgrade a library or any reusable asset) and at the organizational level (*e.g.*, reuse management and reuse monitoring tasks). It is part of APPRAiSER, an environment for visually supporting software reuse tasks with awareness resources [Schots 2014]. Zooming Browser is composed by three core elements (assets, developers, and projects), and the reuse questions are based on the relationships between these elements, as shown in Figure 1 (detailed in [Schots 2014]). While planning its development, some visual metaphors came to mind, but there was no certainty that they were appropriate and whether they would effectively help achieving the established goals. This led to the creation of the mapping structure.

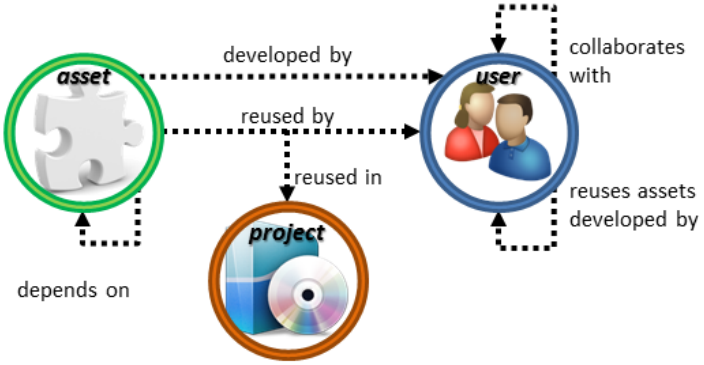


Figure 1. Core elements of Zooming Browser [Schots 2014]

4. The Mapping Structure

The structure for mapping goals and visualizations is presented in Figure 2. All the relationships are many-to-many, except between data and visual attribute, which is one-to-one to avoid user confusion due to ambiguity. During the mapping process, it was noticed that different strategies could take place: *top-down* (when goals are already set and clear), *bottom-up* (when one wants to find the utility of a set of visualizations), *middle-out* (*i.e.*, starting from an intermediary stage towards achievable goals and assisting visualizations), or *meet-in-the-middle* (*i.e.*, when top-down and bottom-up join at some point in their executions). The latter is indicated when there are some goals and visualizations in mind, but some aspects in the middle-part of the mapping are not clear yet and require further reflection and analysis. This was the case of Zooming Browser.

[2] A reuse manager must manage and monitor the overall reuse program, but instead of being merely a managerial role, a technical profile is needed to deal with specificities of assets and the reuse repository.

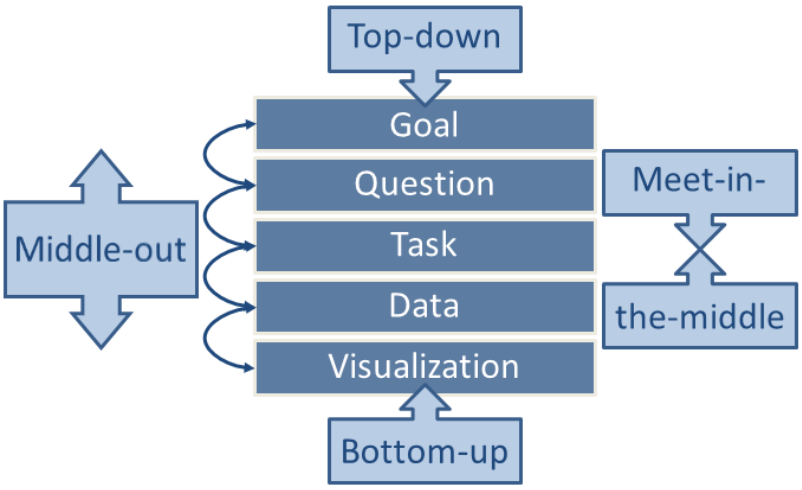


Figure 2. Mapping structure between goals and visualizations

The next subsections present a description of each part of the mapping, with an example of what has been made in the context of Zooming Browser[3], the driver of this work. For didactic purposes, the mapping is described in terms of the top-down strategy.

4.1. Mapping Goals and Questions

The mapping between goals and questions has been thoroughly explored in software engineering [Basili et al. 1994]. It consists of associating questions whose answers help achieving a goal. A question may support more than one goal, and a goal usually consists of more than one question. The GQM format is not mandatory, but is advisable.

For illustration purposes, we present some asset-centric questions[4] derived from one of the Zooming Browser project-related goals (*Decide whether an existing project that already contains a given asset version should upgrade/downgrade to a newer/older asset version*) [Schots 2014]. They are listed as follows. It is advised to keep record of the rationale that relates each question to the goals (this also helps executing the next stage). For instance, *Qc-Qe* point to reuse attempts of a given asset version (which can be successful or not), while *Qf-Qj* provide awareness on the commitment of the asset development team regarding problems identified and features requested, among others.

Qa: *How often is this asset [version] reused over time?*
Qb: *Which consumers reused this asset [version]?*
Qc: *In which projects was this asset [version] reused?*
Qd: *Which projects contain this asset [version] at some point of the development life cycle but do not contain such asset [version] afterwards?*
Qe: *Which projects contain, among their releases, [a version of] this asset?*
Qf: *Which [versions of] assets does this asset [version] depend on?*
Qg: *Among the reported bugs related to this asset [version], how many of them are fixed/open?*
Qh: *How often do producers of this asset fix reported bugs?*
Qi: *How long does it take for producers of this asset to fix reported bugs?*
Qj: *How often do producers of this asset implement improvement suggestions or feature requests?*

[3] The full mapping is not presented in this paper; the parts shown are only for illustration purposes. Zooming Browser is used as example to avoid the risk of misinterpreting a system developed by others.

[4] There are also other questions (including project-centric and developer-centric questions) that are related to this goal, but they are not presented in this paper.

4.2. Mapping Questions and Tasks

One could expect a mapping between questions and metrics, as defined in the GQM approach [Basili et al. 1994]. There is no doubt that metrics can be useful for answering questions, but their interpretation is more intuitive when they are tied to visual representations [Lanza & Marinescu 2006]. Besides, when it comes to interactive visualization tools, it seems more natural to map *questions* to project or organizational *tasks*[5] that must be performed to obtain the answers being sought. It is noteworthy that many questions or tasks are based on literature reports, but it is imperative to assess their relevance to the current state-of-the-practice [Novais et al. 2014]. In the Zooming Browser design, the following tasks are associated with the following questions:

Ta: *Check [the successfulness of] reuse attempts of [a given version of] an asset in existing projects (related to* ***Qa****,* ***Qc****,* ***Qd****,* ***Qe****)*
Tb: *Identify experts (producers/contributors and consumers) on a reusable asset [for communication needs] (related to* ***Qb****)*
Tc: *Understand/Evaluate asset dependencies (related to* ***Qf****)*
Td: *Check if producers have been keeping up with the development of a reused asset (community participation) (related to* ***Qg****,* ***Qh****,* ***Qi****,* ***Qj****)*

4.3. Mapping Tasks and Data

In order to perform software development tasks, it is necessary to resort to data, usually available from different sources. Thus, at this point, one should find out what data are required to support executing such tasks (for the proper identification of relevant data sources and the filtering of unnecessary data). Some processing (data cleaning, integration, aggregation etc.) is usually necessary. Other data may become necessary for complementing the visualization (*e.g.*, due to positioning and organization of data), so it is likely that this stage is revisited afterwards.

Some data for performing the tasks defined for Zooming Browser are listed as follows. They are extracted from reuse repositories, version control repositories, and issue tracking/task manager systems. Links to original sources or other representations (*e.g.*, HTML websites) are also stored for allowing drill-down to additional information.

Project history information *(both from assets' projects and other projects in which assets were reused): project name (related to* ***Ta****,* ***Tb****), commit author (related to* ***Tb****,* ***Td****), commit date (related to* ***Tb****,* ***Td****), added files* (related to* ***Ta****), removed files* (related to* ***Ta****)*
Reuse repository information*: asset name (related to* ***Ta****,* ***Tc****), asset versions (related to* ***Ta****,* ***Tc****), asset dependencies** (related to* ***Tc****)*
Issue tracking information*: issue status (related to* ***Td****), issue type (related to* ***Td****), issue creation date (related to* ***Tb****,* ***Td****), issue close date (related to* ***Tb****,* ***Td****), issue assignee (related to* ***Tb****,* ***Td****), issue resolver (related to* ***Tb****,* ***Td****)*
** Filtering is applied in order to retrieve only commits of assets. ** There are different kinds of software dependencies; the current scope is limited to explicit dependencies (e.g., described in a build configuration file).*

4.4. Mapping Data and Visualizations

This is one of the most important parts of the mapping, because an inappropriate or ambiguous mapping may impair the effectiveness of the visualization tool as a whole. Our studies showed that the mapping between data and visualizations is barely

[5] Interaction tasks with the visualizations (such as filtering, browsing, drill-down etc.) will be handled separately in an upcoming stage.

described in publications, so users have to "guess" it, which can be risky and lead to wrong interpretations of data [Schots & Werner 2014a]. Because this is a more complex and most crucial stage, it can be divided into three different steps, described as follows.

Firstly, one or more visualizations must be already in mind based on the established data and their characteristics; thus, there must be a *pre-selection of candidate visualizations* that will be confirmed later based on the subsequent steps. Secondly, since the visual attributes are responsible for linking data to visualizations, one must *decompose the visual attributes that constitute the visualizations* (such as size, color, position, shape etc.) in order to recognize the data type required by such visual attributes. For instance, different colors enable the representation of categorical data, while color scales require the data type to be continuous. Finally, *mapping each datum to each visual attribute* involves analyzing the available data types to attest their suitability to the visual attributes that compose the visual metaphor. These steps (especially the first two) may require support from skilled visualization experts.

After that, it is possible to ensure that the visual attributes are both necessary and sufficient to the data[6]. Thus, one can be more confident to determine which visualization(s) will be actually used. However, if a data-to-visualization mapping cannot be fully performed, there may be three causes: (i) the available data cannot be mapped to the intended visualization due to incompatibilities (restrictions on data or on visual attributes); (ii) the intended visualization is not sufficient to comprise the necessary data; or (iii) the visualization requires more data than the available ones.

A solution for all these cases can be the choice of different visualizations. An alternative solution for (i) and (ii) is to combine another visualization with the existing one(s) (for instance, through interaction resources or by creating a multi-perspective environment [Carneiro et al. 2010]). In this case, it is important to keep in mind that, ideally, a visual attribute should keep its semantics in a consistent way among different visualizations, in order to avoid misleading interpretations. Finally, for (iii), one may need to collect more data to make the most of a visual metaphor and its resources.

For illustration purposes, an excerpt of the design of the *issues* perspective (from Zooming Browser's *asset-centric* view) is depicted in Figure 3, while Table 1 shows how some of the aforementioned data were mapped to visual attributes of this visualization. Although it is not explicit as a separate stage in the mapping, this stage also requires the choice of interaction resources to be employed, so as to allow users of the visualization tools to explore the data and perform their tasks. In Figure 3, for illustration purposes, issues can be filtered by type through a filtering mechanism.

5. Final Remarks

The mapping structure presented in this paper (and the example of its concrete application) aims at supporting developers of visualization tools in better planning the resources to be used towards the usefulness of these tools, before evaluating them with the intended audience. The mapping stages point out aspects that should be considered

[6] Note that this does not discard the need for performing evaluations (such as usability studies) with the audience of the visualization tools.

when building visualization tools, aiming to make a proper use of the visual attributes of the chosen visualizations, based on the data to represent. This mapping is purposely "open" so that each stage can be accomplished by stakeholders in the most convenient way to them. We hope that this can serve as an initial guidance to future developments of visualization tools, emphasizing the importance of performing each stage carefully.

Figure 3. Issues perspective (draft/sketch)

Table 1. Mapping between data and visual attributes in the issues perspective

Visualization	Visual Attribute	Data	Value	Description
adapted bubble layout	geometric shape	issue	• circle	A circle represents an issue.
	size	issue lifetime	• for closed issues: (*issue close date – issue creation date*) • for open issues: (system current date – *issue creation date*)	The size of an issue is proportional to the time it remains open, *i.e.*, the longer the time an issue has been opened (and not closed) the greater it appears.
	color	issue status	• green, for closed issues • red, for open issues (*issue type* = bug) • yellow, for open issues (*issue type* = feature request or suggestion)	The use of colors helps finding out the status of the issues. Colors provide an overview of how developers are handling issues as they appear.
	icon	issue type	• exclamation mark (*issue type* = bug) • lamp bulb (*issue type* = feature request or suggestion)	Icons facilitate differing issue types. Since bugs are usually more severe or relevant than feature requests, many open bugs may indicate lack of support for the asset's users.

The use of this mapping provided more confidence for implementing Zooming Browser, since we were able to check whether the tool could help answering the established questions before evaluating it with its intended stakeholders. We believe that the effective evaluation of the performed mapping in the context of Zooming Browser may be done by the implementation and use of the tool. This will provide information on what aspects are lacking or could be mapped differently. The evaluation of the tool with its stakeholders is one of the next steps of the research.

We intend to continue this research in order to deepen our understanding with respect to the usefulness of this mapping. Some open questions include: (i) Does this

mapping structure actually support developers of visualization tools in better planning the visualization metaphors and resources to be used? (ii) How can we facilitate each stage of this mapping, *i.e.*, what additional support can be provided for easing such stages (especially the last one)? (iii) Should we build tools to help performing this mapping? We want to investigate with the visualization community answers to these questions and others that may emerge through other applications of the mapping.

Acknowledgment

The authors would like to thank CNPq and FAPERJ, for the financial support for this research, and Natália Schots, for providing useful feedback on this paper.

References

Abuthawabeh, A., Beck, F., Zeckzer, D., Diehl, S. (2013). "Finding structures in multi-type code couplings with node-link and matrix visualizations". In: 1st IEEE Working Conference on Software Visualization (VISSOFT 2013), pp. 1-10, September.

Basili, V., Caldiera, G., Rombach, H. (1994). "Goal Question Metric Paradigm". Encyclopedia of Software Engineering, v. 1, John J. Marciniak, Ed. John Wiley & Sons, pp. 528-532.

Beck, F., Burch, M., Diehl, S. (2013). "Matching application requirements with dynamic graph visualization profiles". In: 17th International Conference on Information Visualisation (IV), London, UK, pp. 11-18, July.

Carneiro, G. F., Sant'Anna, C., Mendonca, M. (2010). "On the Design of a Multi-Perspective Visualization Environment to Enhance Software Comprehension Activities". In: 7th Workshop on Modern Software Maintenance (WMSWM), pp. 61-68, June.

Falconer, S. M., Bull, R. I., Grammel, L., Storey, M.-A. (2009). "Creating Visualizations through Ontology Mapping". In: International Conference on Complex, Intelligent and Software-Intensive Systems (CISIS), pp.688-693, March.

Lanza, M., Marinescu, R. (2006). Object-Oriented Metrics in Practice. Springer-Verlag Berlin Heidelberg New York, 1st ed.

Novais, R., Brito, C., Mendonça, M. (2014). "What Questions Developers Ask During Software Evolution? An Academic Perspective". In: 2nd Workshop on Software Visualization, Evolution, and Maintenance (VEM 2014), pp. 14-21, September.

Schots, M. (2014). "On the use of visualization for supporting software reuse". In: 36th International Conference on Software Engineering (ICSE 2014), Doctoral Symposium, pp. 694-697, June.

Schots, M., Werner, C. (2014a). "Using a Task-Oriented Framework to Characterize Visualization Approaches". In: 2nd IEEE Working Conference on Software Visualization (VISSOFT 2014), pp. 70-74, September.

Schots, M., Werner, C. (2014b). "Characterizing the Implementation of Software Reuse Processes in Brazilian Organizations", Technical Report ES-749/14, COPPE/UFRJ, Rio de Janeiro, Brazil.

Using JavaScript Static Checkers on GitHub Systems: A First Evaluation

Adriano L. Santos[1], **Marco Tulio Valente**[1], **Eduardo Figueiredo**[1]

[1]Department of Computer Sciente – Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos 6627 – 31.270-901 – Minas Gerais – MG – Brazil

{adrianolages,mtov,figueiredo}@dcc.ufmg.br

Abstract. *To improve code quality in JavaScript systems, static code analyzers are used to detect bad coding practices that can be potential bugs and can cause the system to not work properly. This paper investigates bad coding practices in JavaScript reported by two static analyzers (JSHint and JSLint) in order to verify if JavaScript systems are using this lint-like checkers to avoid bad coding practices and what are the most common warnings detected by them. Through an empirical study, we analyze 31 JavaScript systems and our results show that JSHint and JSLint are used by the development team of systems. We also found bad coding practices in all analyzed systems. In five systems, the number of warnings between old and new versions decreased by 14%.*

1. Introduction

Javascript is a widely used client-side language for develop web applications. It is used on various popular websites including Facebook, Gmail, and Twitter. More recently, JavaScript has found its way into server side platforms, such as Node.js[1]. These reasons contribute to a increasing popularity of the language, which became a fundamental piece in modern and interactive Web development [Cantelon et al. 2013].

To avoid pitfalls in JavaScript code, static code analyzers are used to detect problems that may not be verified when a developer is writing code. Static analyzers assist developers looking at code and they can find problems before running it. They inspect code and point some problems based on guidelines [Crockford 2008], [Kovalyov 2015]. In this work we present the results of a study comparing the warnings reported by two popular static code analyzers for JavaScript, called JSHint[2] and JSLint[3]. We observe two categories of warnings reported by these tools: warnings related to bad code logic (Group A) and warnings related to bad code style (Group B). JSHint and JSLint were choosen because they are widely accepted by developers and commonly used in industry.

Warnings of Group A identify pitfalls on code, e.g., mistyped keywords, mistyped variable names, attempts to use variables or functions not defined, use of `eval` function, bad use of operators, etc. These warnings are subtle and difficult to find and can lead to serious bugs. On the other hand, warnings of Group B enforce basic code style consistency rules that do not cause bugs in the code, but make it less readable and maintainable to other developers. This kind of warning is related to lack of proper indentation, lack of curly braces, no semicolon, no use of dot notation, etc.

[1]Node.js. https://nodejs.org/
[2]JSHint. http://jshint.com/.
[3]JSLint. http://www.jslint.com/.

This paper makes the following contributions: (i) We present a list of bad coding practices found by the two tools in an empirical study on 31 JavaScript systems located on GitHub. (ii) We look at whether the warnings indicated by the tools are increasing over a system development time and which warnings are more/less common in the studied systems. (iii) We verify if the systems studied on this present paper are making use of static code analyzers. (iv) We show that the use of static code analyzers are important for improving the code quality.

This paper is structured as follows: Section 2 documents some bad code practices in JavaScript. In Section 3 we detail our methodology to make the empirical study. We also present our results and our findings. Related work is presented in Section 5. Lastly, in Section 6 presents our final conclusions.

2. Bad Coding Practices in JavaScript

The goal of this work is to analyze violations of commonly accepted rules in JavaScript software, as reported by JSHint and JSLint. In this section, we document some of these rules.

Definition (Code quality rule) *A code quality rule is an informal description of a pattern of code or execution behavior that should be avoided or that should be used in a particular way. Following a code quality rule contributes to, for example, increased correctness, maintainability, code readability, or performance* [Gong et al. 2015].

2.1. Using Undeclared Variables

JavaScript allows a variable to be used without a previous declaration. JSHint and JSLint throws a warning when they encounter an identifier that has not been previously declared in a `var` statement or function declaration. In the example of Figure 2.1 we reference the variable `a` before we declare it. In this example both static code analyzers report the warning: `{a} was used before it was defined.`

```
1 function test() {
2     a = 1; // 'a' was used before it was defined.
3     var a;
4     return a;
5 }
```

Fig 1: Code with a warning related to not declared variables.

The warning displayed in Figure 1 is raised to highlight potentially dangerous code. The code may run without error, depending on the identifier in question, but it is likely to cause confusion to other developers and the piece of code could in some cases cause a fatal error that will prevent the rest of script from executing.

2.2. Shadowing variables

Variable shadowing occurs when a variable declared within a certain scope (decision block or function) has the same name as a variable declared in an outer scope. This kind of problem generates `{a} is already defined` warning.

```
1 var currencySymbol = "$"; //This variable is shadowed by inner function variable.
2
3 function showMoney(amount) {
4   var currencySymbol = "R$"; // '{currencySymbol}' is already defined.
5   document.write(currencySymbol + amount); //R\$ sign will be shown.
6 }
7 showMoney("100");
```

Fig 2: Code with a warning related to shadowing variable.

In the example of Figure 2 the variable `currencySymbol` will retain the value of the final assignment. In this case, a `R$` sign will be shown, and not a dollar, because the `currencySymbol` containing the dollar is at a wider (global) scope than the `currencySymbol` containing the `R$` sign. This type of error can occur because the programmer can mistyped the identifier of one of the variables. Renaming one of them can solve this problem.

2.3. Mistakes in conditional expressions

Sometimes programmers forgotten to type an operator and instead of defining a conditional expression they define an assignment to a variable. JSHint and JSLint reports a warning called `expected a conditional expression and instead i saw an assignment` when this situation occur, as illustrated in Figure 3.

```
1  function test(someThing) {
2      do {
3          someThing.height = '100px';
4      } while (someThing = someThing.parentThing); //expected a
5      // conditional expression and instead i saw an assignment
6          ...
7  }
```

Fig 3: Code with a warning related to confusing conditional expression.

3. Study

The JavaScript systems considered in this study are available at GitHub. We selected systems ranked with at least 1,000 stars at GitHub, whose sole language is JavaScript, and that have at least 150 commits. This search was performed on May, 2015 and resulted in 31 systems from different domains, covering frameworks, editors, games, etc [Silva et al. 2015]. After the checkout of each system, we manually inspected the source code to remove the following files: compacted files used in production to reduce network bandwidth consumption (which have the extension *.min.js), documentation files (located in directories called doc or docs), files belonging to third party libraries, examples and test files.

The selected systems are presented in Table 1, including their version, size (lines of code), LOC/#Total warnings detect by JSHint, LOC/# Total warnings detected by JSLint, number of warnings from Group A and Group B reported by JSHint and JSHint and the total number of warnings detected by the two static analyzers. We use Eclipse with JSHint plug-in (version 0.9.10) and JSLint plug-in (version 1.0.1) to analyze the systems in Table 1. In our study, we used the default settings of each plugin.

Table 1 presents the total number of warnings detected by JSHint and JSLint. Both static analyzers found pitfalls and code style warnings in all systems (100%). The number of warnings for each system in Table 1 is the total number of warnings found including repetitions of a same type of warning. JSHint and JSLint found 52 different types of warnings in the studied systems. The system with the largest number of warnings is `mocha` (2401 warnings), followed by `babel` (2328 warnings) and `wysihtml5` (2303 warnings). This high number of warnings correspond to warnings of group B (code style warnings). Figure 4 show the total warnings encountered by each static analyzers. The total number of warnings found by JSHint was 20.12% for Group A and 79.88% of Group B. JSLint reports 6.24% and 93.76% warnings of Group A and B, respectively. Furthermore, columns 4 and 5 of Table 1 show the density of warnings encountered by each static analyzer in the analyzed systems. For example, for `gulp`, a warning is found by JSHint for each 4.27 lines

Table 1. JavaScript systems (ordered on the LOC column).

System	Version	LOC	LOC/#Total JSHint	LOC/#Total JSLint	#Warnings Group A JSHint	#Warnings Group A JSLint	#Warnings Group B JSHint	#Warnings Group B JSLint	#Total JSHint	#Total JSLint
masonry	3.1.5	197	16.47	1.11	0	12	12	165	12	177
gulp	3.7.0	282	4.27	3.66	17	18	49	59	66	77
randomColor	0.1.1	361	30.08	7.22	7	11	5	39	12	50
respond	1.4.2	460	0	3.40	0	2	0	133	0	135
mustache.js	0.8.2	571	71.37	12.14	8	5	0	42	8	47
clumsy-Bird	0.1.0	628	11.48	4.21	3	8	52	141	55	149
deck.js	1.1.0	732	183	14.93	1	14	3	35	4	49
impress.js	0.5.3	769	0	40.47	0	0	0	19	0	19
isomer	0.2.4	770	13.27	1.75	3	26	55	414	58	440
fastClick	1.0.2	798	0	16.62	0	0	0	48	0	48
parallax	2.1.3	1007	0	968	0	7	0	97	0	104
intro.js	0.9.0	1026	20.52	21.37	32	5	18	43	50	48
alertify.js	0.5.0	1036	20.72	21.37	28	3	22	146	50	149
async	0.9.0	1117	21.48	15.30	3	5	49	68	52	73
socket.io	1.0.4	1223	101.91	13.89	7	5	5	83	12	88
qunit	1.14.0	1379	197	26.01	5	3	2	50	7	53
underscore	1.6.0	1390	43.43	46.33	30	6	2	24	32	30
slick	1.3.6	1684	842	42.41	2	1	0	39	2	40
turn.js	3.0.0	1914	34.17	20.80	26	2	30	90	56	92
numbers.js	0.4.0	2454	175	15.24	5	27	9	134	14	161
cubism.js	1.6.0	2456	16.26	5.62	61	81	90	356	151	437
zepto	1.1.6	2456	4.10	4.54	54	32	545	508	599	540
typeahead.js	0.10.2	2468	32.90	2.87	55	14	20	844	75	858
rickshaw	1.5.1	2752	12.41	1.83	31	65	190	1433	221	1498
express	4.4.1	2942	50.72	22.98	14	7	44	121	58	128
knockout	3.3.0	3157	4.23	3.35	96	81	649	861	745	942
jasmine	2.3.4	3956	44.44	13.18	21	28	68	272	89	300
mocha	2.2.5	4225	3.76	3.30	98	30	1025	1248	1123	1278
slickgrid	2.1.0	5345	15.44	3.56	195	44	151	1456	346	1500
babel	5.4.7	5893	4.31	6.11	167	150	1198	813	1365	963
wysihtml5	0.3.0	5913	20.11	2.94	149	88	145	1921	294	2009

of code. JSLint finds a warning for each 3.66 lines of code. The low number of Group A warnings (warnings that can cause the system to do not function properly) indicates that programmers are cautious and want to ensure code quality and they may be making use of static analyzers in their systems. In `impress.js` system, only 19 warnings of Group B were found by JSLint. Another system which showed only warnings of Group B was `fastclick` (48 warnings found by JSLint). In all systems, JSHint detected less warnings that JSLint. On the other hand, JSHint detected more warnings of group A that JSLint (20.12% versus 6.24% respectively).

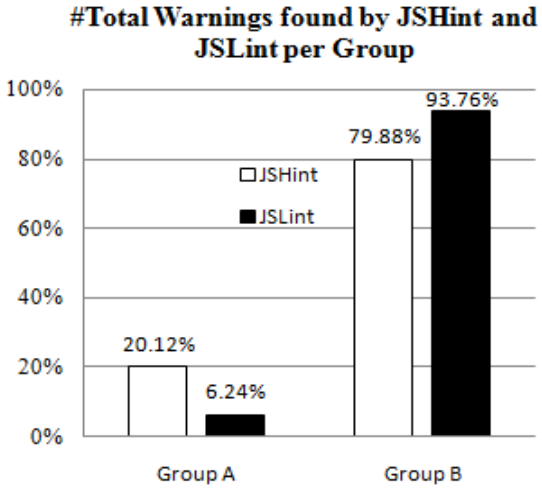


Figure 4. Percentage of warnings detected by JSHint and JSLint.

Table 2. Type of warnings found (ordered by #Systems column).

Warning	#Systems		Warning	#Systems		Warning	#Systems	
	JSHint	JSLint		JSHint	JSLint		JSHint	JSLint
{a} is already defined ∗	12	0	{a} is out of scope ∗	7	1	missing semicolon †	18	1
use '!==' to compare with '0', 'null', or 'undefined' ∗	9	10	unnecessary semicolon †	6	1	use '===' to compare with '0', 'null', or 'undefined' ∗	8	9
use 'isNaN' function instead compare with 'NaN' ∗	1	0	eval is evil ∗	4	3	missing Break Statement before 'case' ∗	2	1
expected a conditional expression and instead i saw an assingment ∗	8	0	expected an assignment and instead i saw an expression ∗	17	2	don't make functions within a loop ∗	11	4
missing '()' invoking a constructor ∗	11	5	{a} was used before it was defined ∗	3	3	do not use 'new' for side effects ∗	0	5
missing 'new' prefix when invoking a constructor ∗	1	2	expected {a} at column x, not column y †	0	20	possible Strict Violation ∗	2	0
missing name in function declaration ∗	1	0	unexpected space after †{a}	0	2	unnecessary Semicolon †	6	1
do not declare variables in a loop †	0	4	bad line breaking before {a} †	10	0	read only ∗	1	2
variable was not declared correctly ∗	3	2	combine this with the previous var statement ∗	0	8	empty block †	0	9
expected an identifier and instead saw 'undefined' (a reserved word) ∗	1	8	missing "use strict" statement †	0	22	use the function form of 'use strict' †	2	3
move "var" declarations to the top of the function †	0	14	unexpected dangling '_' in {a} †	0	17	weird relation ∗	0	1
weird condition ∗	0	1	Script URL ∗	2	0	[{a}] is better written in dot notation †	6	7
did you mean to return a conditional instead of an assignment? ∗	2	0	do not use {a} as a constructor ∗	1	0	don't use 'with' ∗	1	0
the '_proto_' property is deprecated †	4	2	unexpected ++ †	0	12	constructor name should be start with a uppercase letter †	0	3
unreacheable {a} after 'throw', 'return' ∗	3	0	comma warnings can be turned off with 'laxcomma' †	3	0	unexpected sync method ∗	0	6
unexpected TODO Comment †	0	3	missing name in function statement ∗	0	1	move the invocation into the parens that contain the function ∗	0	5
missing space after {a} †	0	9	unexpected 'typeof'. Use '===' to compare directly with undefined ∗	0	6			
use spaces not tabs †	0	6	Unexpected 'in'. Compare with undefined, or use the hasOwnProperty method instead †	0	3			

Table 2 presents the number of systems who have occurrences of a particular type of warning. For example, the warning (`{a}is already defined`) appears in twelve systems and was detected only by JSHint. Some warnings displayed by JSHint and JSLint have the same meaning, but have different names. For example, warnings about use of `eval` function. JSHint shows a warning with the text: `eval can be harmful` and JSLint shows the text: `eval is evil`. In this study, we consider the terminology used by JSLint. In the analyzed systems 52 types of warnings were found. JSHint and JSLint are able to detect more than 150 types of warnings [Kovalyov 2015]. Warnings marked with ∗ belong to Group A and warnings marked with † belong to Group B.

The most common warnings shown in Table 2 are:

- `missing "use strict" statement` (occurred in 22 systems), this warning is raised to highlight a deviation from the strict form of JavaScript coding. This warning can be helpful as it should highlight areas of code that may not work as expected, or may even cause fatal JavaScript errors.
- `Expected {a} at column x, not column y` (occurred in 20 systems), this code style warning (group B) warns programmers about lack of indentation.
- `Unexpected dangling '_' in {a}` (occurred in 17 systems), this warning reported by JSLint warns about use of underscore character in variables name.
- `Move "var" declarations to the top of the function` (occurred in 14 systems), this warning is thrown when JSLint encounters a variable declaration in a `for` or `for-in` statement initializer.
- `{a} is already defined` (occurred in 12 systems), only this warning belong to group A and and was detected only by JSHint.

The less common warnings shown in Table 2 are:

- `do not use {a} as a constructor` (1 ocurrence), this warning is thrown when JSHint or JSHint encounters a call to `String`, `Number`, `Boolean`, `Math` or `JSON`

preceded by the `new` operator. This kind of bad practice can become a fatal error in JavaScript.

- `Weird relation` (1 occurrence), this warning is raised when JSLint encounters a comparison in which the left hand side and right hand side are the same, e.g. `if (x === x) or if ("10" === 10).`
- `missing name in function declaration` (1 ocurrence).
- `use 'isNaN'function instead compare with 'NaN'` (1 occurrence).
- `do not use {a} as constructor` (1 occurrence).

We also investigated whether warnings detected by two static analyzers increase or decrease along the different versions of the analyzed systems. We selected five systems with more number of warnings of all groups and analyze an earlier version with at least one year of difference between the versions. Figure 5 shows the total number of warnings reported by JSHint and JSLint for each system in an earlier and previous version. For example, the wysihtml5 system in version 0.2.0 presents 335 warnings detected by JSHint and 2290 warnings detected by JSLint. The version 0.3.0 of wysihtml system which has a year of difference to version 0.2.0 presents 294 and 2009 warnings detected by JSHint and JSLint respectively, a reduction of 14% of warnings. In the five systems analyzed in Figure 5 there was reduction of warnings. On average the reduction was 14% for JSHint and 12% for JSLint. These results may indicate that developers are using static checkers, as JavaScript IDE's have JSHint and JSLint embedded.

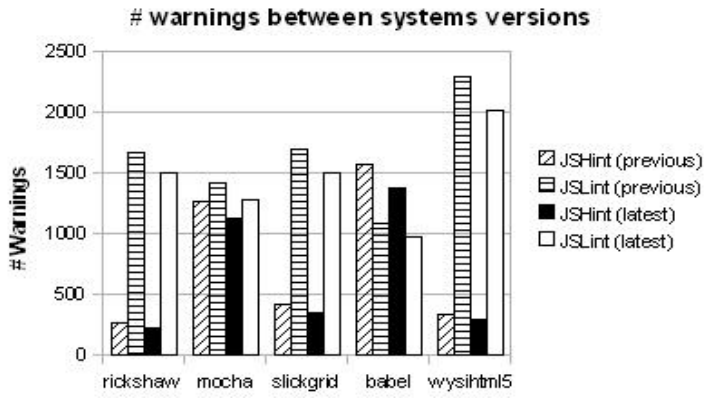


Figure 5. Warnings detected by JSHint and JSLint in different systems versions.

Threats to Validity: Our systems sample can be small and not be representative. We minimize this threat by selecting projects of different sizes of LOC and different domains. Static analyzers may have incorrectly considered some code snippets as bad coding practices. During the analysis of systems, we minimize this kind of threat, by manual inspection of some code snippets and making sure there were no bad coding practices.

4. Evaluation with Developers

We chose fifteen systems with more warnings and report the results to the team of developers through pull-requests on GitHub. We obtained responses from 11 development teams. For the SlickGrid system, the development team said that use JavaScript code editing tools that already has an integrated static analyzer and that many of the errors found by our study are issues related to style and organization of code (Group B) and they modify the analyzer to not detect such problems. In two other systems, Jasmine and Knockout developers use JSHint with added specific rules for their projects. In addition, developers

of Jasmine and Knockout systems will analyze the version tested by our study and examine the problems founded. In seven systems, developers said they are aware the warnings can become potential bugs, but they will not fix these problems right now because there are other priorities in the project. In addition, the developers of these seven systems said that many errors can be related to environment variables of third-party libraries that can be used in the project code without being declared, for example, variables and functions of the jQuery library. The developers of randomColor system not using said static code analyzers because the system has fewer lines of code (LOC 361).

According to the answers of the developers we realize that for large projects static analyzers are used. Futhermore, developers are more concerned with warnings of Group A (which can become potential bugs) than with warnings of Group B (stylistic problems). One developer answered that static code analyzers do not understand more advanced syntax of JavaScript properly. In addition, the developer suggested that static analyzers should have an option for users to be able to inform their level of knowledge in JavaScript.

"I use WebStorm for JavaScript editing, which has the lint/hint tools built in. I checked it out and there were indeed about 10 reasonably serious errors it picked up. However a lot of warnings these tools flag are the result of them not understanding the syntax of more advanced JavaScript properly. So they are useful to a point, but can be quite annoying when you have to trawl through a bunch of incorrectly flagged errors to find the one true error. This is what tends to put people off. I notice also that some errors being flagged in my project are in the jQuery and other libraries. This is a sign in itself. These libraries tend to use nonstandard but highly optimised ('guru level') code structures. It would perhaps be good if the Lint user could flag their level of expertise (newbie, competent, guru) to tell the tool that suspicious but correct looking code is probably not a mistake but the work of a guru, or to at least allow different types of structure in this case."

5. Related Work

Some works in the literature use JSHint and JSLint as auxiliary tools to check bad practices of JavaScript coding, but so we could not find any work comparing static analyzers of JavaScript code. DLint [Gong et al. 2015] presents a dynamic tool for checking bad coding practices in JavaScript and concludes that JSHint is a good checker and must be used in addiction with dynamic checkers. In this study they found that static tools report more warnings of Group B than of Group A and group B warnings are easier to solve. JSNOSE [Fard and Mesbah 2013] presents a technique for detecting code smells that combines static and dynamic analysis in order to find patterns in the code that could result in understanding and maintenance problems. [Politz et al. 2011] use a novel type system for JavaScript to encode and verify sandboxing properties, where JSLint is used in the verification process. FLIP Learning [Karkalas and Gutierrez-Santos 2014] present an Exploratory Learning Environment (ELE) for teaching elementary programming to beginners using JavaScript. They used rules addressed by JSLint to help programmers. [Horváth and Menyhárt 2014] also use static code analyzers to teach introductory programming with JavaScript in higher education.

6. Conclusion

In this study, we found that even in notably known JavaScript systems bad coding practices are found by JSHint and JSLint. We noticed that systems have a low amount of

warnings due to the use of static code analyzers, since these tools are able to detect more than 150 different types of pitfalls. Besides, our study shows that both tools detected problems in groups A and B, and JSHint detected a greater number of warnings from Group A (20.12%) in the systems and JSLint a greater number of warnings of group B (93.76%). But that is not enough to say which tool has best rate to find bad coding practices since their results are similar in this study. Furthermore, according to the response of developers we proved that 10 of the 31 analyzed systems static analyzers are used. For future works the field of study for static analyzers for JavaScript code is large and there are several approaches that can be exploited as an analysis of which bugs that are fixed on JavaScript systems are related to the warnings pointed out by JSHint and JSLint tools. In addition, we can improve the static analyzers according to our results, improving other lint-like analyzers to find a larger number of warnings that can become potential bugs and that are more subtle to find in a naive inspection of source code. We also plan to evaluate the false positives raised by JSHint and JSLint following a methodology used in the evaluation of Java static checkers [Araujo et al. 2011].

Acknowledgments: This research is supported by FAPEMIG and CNPq.

References

Araujo, J. E., Souza, S., and Valente, M. T. (2011). A study on the relevance of the warnings reported by Java bug finding tools. *IET Software*, 5(4):366–374.

Cantelon, M., Harter, M., Holowaychuk, T., and Rajlich, N. (2013). *Node.Js in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Crockford, D. (2008). *JavaScript - the good parts: unearthing the excellence in JavaScript.* O'Reilly.

Fard, A. and Mesbah, A. (2013). JSNOSE: Detecting JavaScript Code Smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125.

Gong, L., Pradel, M., Sridharan, M., and Sen, K. (2015). Dlint: Dynamically checking bad coding practices in javascript. In *ISSTA - International Symposium on Software Testing and Analysis*, pages 94–105.

Horváth, G. and Menyhárt, L. (2014). Teaching introductory programming with javascript in higher education. In *Proceedings of the 9th International Conference on Applied Informatics*, pages 339–350.

Karkalas, S. and Gutierrez-Santos, S. (2014). Enhanced javascript learning using code quality tools and a rule-based system in the flip exploratory learning environment. In *Advanced Learning Technologies (ICALT), 2014 IEEE 14th International Conference on*, pages 84–88.

Kovalyov, A. (2015). *Beautiful JavaScript: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., 1st edition.

Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. (2011). Adsafety: Type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA. USENIX Association.

Silva, L., Ramos, M., Valente, M. T., Anquetil, N., and Bergel, A. (2015). Does JavaScript software embrace classes? In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 73–82.

Avaliação Experimental da Relação entre Coesão e o Esforço de Compreensão de Programas: Um Estudo Preliminar

Elienai B. Batista[1], Claudio Sant'Anna[1]

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

elienaibittencourt@gmail.com, santanna@dcc.ufba.br

Abstract. *The software engineering literature claims that cohesion is a software design attribute that influences program comprehensibility. Researchers have defined a series of metrics for quantifying cohesion. However, there is no empirical evidence about whether there is a relation between cohesion, quantified by means of metrics, and program comprehension effort. In this paper, we present a quasi-experiment that evaluates the relation between cohesion measures and the effort for understanding the source code of object-oriented systems' classes.*

Resumo. *A literatura de engenharia de software afirma que coesão é um atributo de design do software que influencia a compreensibilidade de programas. Pesquisadores definiram uma série de métricas para quantificar coesão. No entanto, faltam evidências experimentais sobre a existência de uma relação entre coesão, quantificada por meio de métricas, e o esforço para se compreender programas. Nesse artigo, apresentamos um quase-experimento que avalia a relação entre medidas de coesão e o esforço para se compreender o código fonte de classes de sistemas orientados a objetos.*

1. Introdução

Coesão é um conceito bem conhecido e usado como atributo de qualidade interna de *design* do software. A coesão de um módulo corresponde ao grau pelo qual o módulo se dedica a implementar apenas uma responsabilidade do sistema [Pflegger and Atlee 2010]. Afirma-se que um software bem projetado tem os módulos bem coesos. Afirma-se também que o grau de coesão dos módulos de um sistema pode influenciar atributos de qualidade externa importantes, como manutenibilidade, facilidade de compreensão e facilidade de reutilização [Pflegger and Atlee 2010].

Várias métricas tem sido definidas e utilizadas para quantificar valores de coesão de software orientado a objetos [Chidamber & Kemerer 1994; Henderson- Sellers et al. 1996; Briand et al. 1998; Silva et al. 2012]. A maioria dessas métricas quantifica coesão de cada uma das classes que compõem o código fonte do sistema. Essas métricas são de dois tipos: estruturais e conceituais. As métricas estruturais baseiam-se nas relações de dependência estrutural entre os métodos da classe para quantificar coesão [Briand et al. 1998]. Esse tipo de métrica considera que uma classe é bem coesa se a maioria dos seus métodos dependem um do outro ou acessam pelo menos um mesmo atributo da classe. Por outro lado, as métricas conceituais baseiam-se em informações dos conceitos implementados pelos métodos e nas relações de dependência conceitual entre eles. Se a maioria dos métodos implementam os mesmos conceitos, a classe é considerada bem

coesa. A maioria das métricas conceituais usa técnicas de mineração de textos para determinar os conceitos implementados por cada método [Marcus & Poshyvanyk 2005; Silva et al. 2012].

Acredita-se que, quanto maior for coesão, menor pode ser o esforço para se compreender um programa [Pflegger and Atlee 2010]. Módulos com baixa coesão são, teoricamente, mais difíceis de compreender, pois possuem código fonte relativo a diferentes responsabilidades, o que pode atrapalhar o entendimento de cada uma delas.

Compreensão de programa consiste da realização de atividades para se obter conhecimento geral sobre o código fonte de um sistema, as funcionalidades que ele implementa e como ele está estruturado [Bois et al. 2006]. Estima-se que desenvolvedores dediquem em média mais da metade do esforço de manutenção de software com atividades de compreensão [Rugaber 1995]. Além disso, parte substancial do esforço de compreensão de um sistema está relacionada à compreensão do seu código fonte.

Apesar de existir uma série de métricas de coesão e de se acreditar que coesão influencia a compreensibilidade de programas, poucos estudos foram realizados para avaliar qual é a relação entre coesão, quantificada por meio de métricas, e o esforço para se compreender o código fonte de sistemas de software. Diante desse contexto, realizamos um estudo preliminar com o objetivo de avaliar experimentalmente em que nível o grau de coesão de classes de sistemas de software orientados a objetos está relacionado ao esforço para compreender seu código fonte. Avaliamos também se os diferentes tipos de métricas – estrutural e conceitual – tem relação diferente com o esforço de compreensão. No estudo, participantes executaram tarefas de compreensão do código fonte de classes com diferentes graus de coesão conceitual e estrutural, quantificados por meio de métricas de código fonte. Os resultados trouxeram evidência da dificuldade de se analisar isoladamente o impacto de coesão, um vez que outros atributos do código fonte também podem influenciar a compreensibilidade. Por causa disso, os resultados não forneceram evidências conclusivas a respeito da relação entre coesão e os esforço para se compreender programas. Por outro lado, o design do estudo se mostrou promissor, principalmente no que tange a medição do esforço de compreensão de programas.

2. Trabalhos Relacionados

Os trabalhos relacionados, em sua maioria, ou (i) dedicam-se a avaliar a relação entre outras características do código fonte e compreensibilidade de programa, como Feigenspan et al. (2011), ou (ii) estudam a relação entre coesão com outros atributos de qualidade, como a tendência a mudanças ou a tendência a defeitos, como Silva et al. (2012). Não encontramos nenhum trabalho que avalie explicitamente se medidas de coesão estão associadas ao esforço para se compreender programas.

Feigenspan et al. (2011), por exemplo, analisam se e em que grau medidas de complexidade e tamanho, por exemplo, número de linhas código, estão relacionadas a compreensão do programa. Seus resultados mostram que não há correlação entre essas características do código fonte e a compreensão do programa. Os autores acreditam que possa existir uma relação entre coesão e acoplamento com a compreensão de programa e por isso recomendam uma investigação dessa relação. Silva et al. (2012), por outro lado, analisam a relação entre coesão com a tendência que as classes tem de sofrer mudanças. Eles avaliam a correlação estatística entre uma série de métricas de coesão e o número de

revisões (commits) que classes sofreram ao longo da evolução de alguns sistemas. Os resultados evidenciaram a existência de uma correlação positiva moderada entre algumas métricas de coesão e o número de revisões das classes.

3. Design do Estudo

O estudo experimental teve como objetivo responder as seguintes questões de pesquisa: **Questão 1:** A coesão de uma classe apresenta alguma influência sobre a compreensão do seu código?

Questão 2: Há diferença no esforço de compreensão de classes com diferentes valores de coesão conceitual e coesão estrutural?

O estudo teve a seguinte configuração: No laboratório, dezoito participantes realizaram atividades que demandaram a compreensão do código fonte de quatro classes. A realização dessas atividades permitiu que medíssemos o esforço desprendido por cada participante para compreender cada classe. As classes tinham diferentes valores de coesão para que pudéssemos compará-los com o esforço de compreensão. O estudo se caracterizou por ser preliminar devido pequena quantidade de participantes e classes envolvidas. Além disso, podemos classificar o estudo como um quase-experimento, pois os participantes foram selecionados por conveniência. A seguir mais detalhes do design do estudo são descritos.

Métricas: Utilizamos a métrica de coesão estrutural *Lack of Cohesion in Methods 5* (LCOM5) [Henderson- Sellers et al. 1996] e a métrica de coesão conceitual *Lack of Concern-Based Cohesion* (LCbC) [Silva et al. 2012]. Selecionamos LCOM5 por se tratar da versão mais recente da tradicional métrica de coesão LCOM [Chidamber & Kemerer 1994]. Para quantificar coesão, LCOM5 considera a dependência entre métodos por meio do acesso a atributos em comum. Quanto maior a quantidade de métodos que acessam atributos em comum, maior é a coesão da classe e vice-versa. Os valores de LCOM5 variam de zero a um. Quanto menor o valor de LCOM5 mais coesa é a classe. Portanto, se uma classe tem valor de LCOM5 igual a zero significa que todos os seus métodos acessam os mesmo atributos e, portanto, são bem coesos entre si. No estudo, calculamos os valores de LCOM5 por meio da ferramenta *Metrics*[1].

Selecionamos a métrica conceitual LCbC por ela já ter sido usada em outros estudos que a comparam com métricas estruturais [Silva et al. 2012; Silva et al. 2014]. LCbC conta o número de interesses presentes em cada classe [Silva et al. 2012]. Interesses são conceitos implementados em um sistema, como requisitos, funcionalidades e regras de negócio. Quanto mais interesses uma classe implementa, menos coesa ela é, e vice-versa. Para quantificar essa métrica é preciso determinar quais interesses cada método de uma classe implementa [Silva et al. 2012]. Isso pode ser feito manualmente ou usando alguma técnica de mineração de texto, como acontece com outras métricas conceituais [Liu, Y. et al]. Devido o pequeno número de classes, foi possível determinar manualmente os valores de LCbC para as classes do estudo.

Seleção de Classes: Essa foi a etapa mais crítica do estudo, pois ao selecionar as classes teríamos que minimizar ao máximo a influência de fatores que também pudessem influenciar coesão (fatores de confusão). Selecionamos, portanto, classes com as

[1]http://metrics.sourceforge.net/

[2] http://guimteixeira.wordpress.com/2010/05/16/exercicio-de-java-agenda-telefonica/

seguintes características: (i) tamanhos semelhantes em termos do número de linhas de código, (ii) domínio simples e acessível a todos os participantes, (iii) nomenclatura de identificadores com qualidade semelhante, (iv) e ausência de comentários. Além desses fatores, outro aspecto importante para a seleção das classes foram os valores das métricas LCOM5 e LCbC. A Tabela 1 apresenta os valores de coesão das classes selecionadas. Para permitir a comparação entre diferentes valores de coesão estrutural e conceitual, selecionamos: (i) uma classe com alta coesão estrutural e alta coesão conceitual (Contatos[2]), (ii) uma classe com baixa coesão estrutural e baixa coesão conceitual (Locadora2[3]), (iii) uma classe com alta coesão estrutural e baixa coesão conceitual (BibliotecaUI2[4]) e (iv) uma classe com baixa coesão estrutural e alta coesão conceitual (Person2[5]).

Tabela 1. Valores das métricas das classes selecionadas

Métrica	BibliotecaUI2	Person2	Locadora2	Contatos
LCOM5	0	0,93	0,6	0
LCbC	4	1	4	1
Número de Linhas (LOC)	274	254	240	200

Medição do esforço de compreensão: Para quantificar o esforço necessário para compreender cada classe usamos duas métricas: (i) o tempo utilizado pelos participantes para responder quatro perguntas sobre cada uma das classes e (ii) o número de perguntas com respostas erradas. As perguntas apresentadas nos questionários exigiam uma simulação mental do código fonte, como por exemplo, "Se o método *m* receber como entrada os valores *x* e *y*, qual será seu retorno?" Essa estratégia está alinhada com o que a literatura recomenda para se medir esforço de compreensão [Bois et al. 2006]. Segundo Dunsmore and Roper (2013) esse tipo de pergunta reflete melhor a compreensão do participante acerca do código fonte.

Participantes: A amostra foi composta por 18 participantes: 14 alunos de graduação e quatro alunos de pós-graduação em Ciência da Computação da Universidade Federal da Bahia. Aplicamos um questionário para verificar a experiência dos participante. De acordo com as respostas, consideramos sete participantes como experientes (mais de três anos trabalhando com programação) e 11 pouco experientes.

Estudo Piloto: Para avaliar a configuração do experimento realizamos um estudo piloto com dois participantes, um experiente e outro pouco experiente. Por meio do estudo piloto, verificamos que a execução do experimento duraria em média 50 minutos. Além disso, corrigimos alguns erros ortográficos encontrados nos questionários e verificamos que alguns esclarecimentos deveriam ser comunicados aos participantes antes da execução do experimento.

3.1. Ameaças à Validade

Abaixo apresentamos possíveis ameaças à validade do experimento e as ações realizadas para minimizá-las.

---

[2] http://guimteixeira.wordpress.com/2010/05/16/exercicio-de-java-agenda-telefonica/

[3] http://ifpr2011.blogspot.com.br/2011/09/locadora-em-java-usando-modo-texto.html

[4] http://www.robsonmartins.com/inform/java/persistencia.php

[5] http://www.soberit.tkk.fi/mmantyla/ISESE2006/

Validade Interna: A fadiga dos participantes ao analisar o código fonte é uma possível ameaça à validade interna. Para minimizar a influência da fadiga, configuramos o estudo de forma que sua execução não ultrapassasse uma hora de duração. Participantes com diferentes níveis de experiência em programação poderiam ser outra ameaça. Verificamos a experiência de cada participante, por meio de um questionário de caracterização, e observamos que a diferenças em graus de experiência não afetaram os resultados. Também nos preocupamos com o fato da falta de similaridade entre as classes poder influenciar os resultados. Tentamos minimizar essa ameaça levando em consideração alguns fatores de confusão ao selecionar as classes.

Validade Externa: O estudo possui a limitação de não poder ser generalizado devido à pequena quantidade de classes e à pequena amostra de participantes, composta somente por estudantes. Outra ameaça está relacionada ao fato dos participantes terem usado um editor de texto (WinEdit) mais limitado que os ambientes de programação comumente usados em ambientes profissionais. Fizemos essa opção de propósito, pois o WinEdit não fornece notificações de erros no código nem sugere soluções. Como existia uma pergunta que pedia para o participante completar uma linha em branco do código, não queríamos que ele tivesse ajuda do editor.

Validade de Construto: A clareza das perguntas apresentadas aos participantes poderia representar uma ameaça, uma vez que diferentes participantes poderiam interpretar uma mesma pergunta de maneiras distintas. Por isso, avaliamos e ajustamos as perguntas com base no estudo piloto. A forma como medimos o esforço de compreensão poderia ser outra ameaça, pois se trata de algo subjetivo. Para isso, utilizamos procedimentos indicados pela literatura e já utilizadas em outros estudos.

4. Resultados e Discussões

A Figura 1 mostra o tempo médio para analisar e responder as perguntas relativas a cada classe. O tempo médio foi calculado somando-se o tempo que os participantes levaram para responder as quatro perguntas de cada classe e dividindo pelo número de participantes. Podemos observar que a classe Person2 foi a que demandou mais tempo de análise: 22 minutos. Essa classe possui baixa coesão estrutural (LCOM5 = 0,93) e alta coesão conceitual (LCbC = 1). É importante destacar que, apesar de termos selecionados classes com tamanhos similares, não foi possível encontrar classes com exatamente o mesmo número de linhas de código. Porém, Person2 não é a maior classe. Ela tem LOC = 254, enquanto BibliotecaUI2 tem LOC = 274 e é a maior classe.

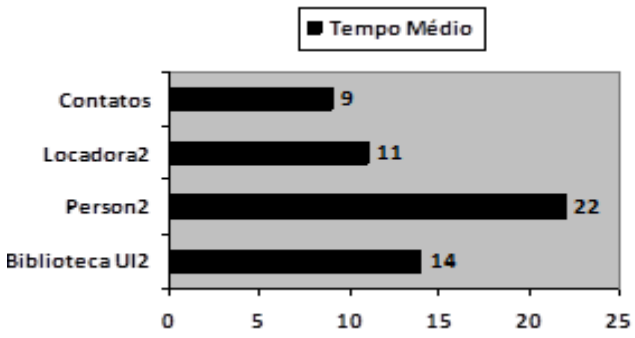


Figura 1. Tempo médio para analisar cada classe, em minutos.

BibliotecaUI2 demandou o segundo maior tempo para ser analisada: 14 minutos. Mesmo assim esse tempo não é muito maior que os tempos de análise requeridos pelas demais classes: Locadora2 (11 minutos) e Contatos (9 minutos). Em termos de coesão, BibliotecaUI2 apresenta características inversas a Person2: alta coesão estrutural (LCOM5 = 0) e baixa coesão conceitual (LCbC = 4).

A classe Contatos é a mais bem coesa tanto conceitualmente (LCbC = 1) quanto estruturalmente (LCOM5 = 0). Por outro lado, a classe Locadora2 apresenta baixa coesão conceitual (LCbC = 4) e baixa coesão estrutural (LCOM5 = 0,6). Apesar do contraste em relação aos valores de coesão, as duas classes requereram os dois menores tempo de análise: 9 minutos para Contatos e 11 minutos para Locadora2.

A Figura 2 mostra o número médio de erros cometidos pelos participantes durante a análise de cada classe. Um erro significa que o participante forneceu resposta errada para uma das questões relacionadas à classe. O número médio de erros por classe é a soma do número de erros para a classe dividido pelo número de participantes.

Os resultados observados na Figura 2 são bem semelhantes aos resultados do tempo médio (Figura 1). Considerando tanto o tempo médio quanto o número médio de erros, a classe Person2 foi a classe que demandou maior esforço para ser compreendida enquanto a classe Contatos demandou o menor esforço. Isso aumenta a confiança do estudo, pois demonstra que não houve casos em que o participante cometeu menos erros em uma classe porque gastou mais tempo analisando-a.

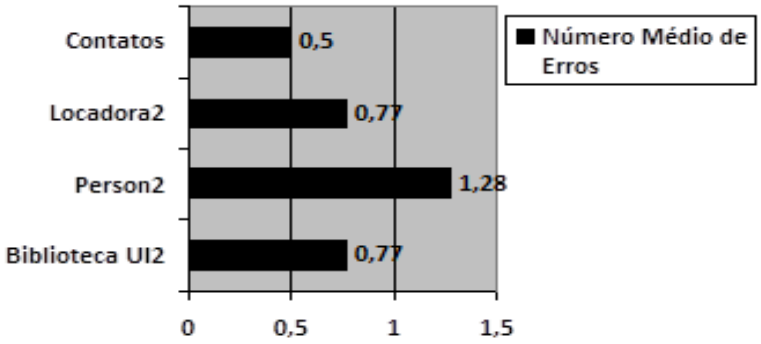


Figura 2. Número médio de erros de cada classe.

Analisando dados sob a ótica do perfil dos participantes, observamos que os sujeitos considerados experientes tiveram melhor desempenho. Para esse grupo o maior número de erros obtidos foi quatro e o maior tempo de análise foi de 68 minutos. Para o grupo considerado pouco experiente o maior número de erros foi oito e o maior tempo gasto foi de 81 minutos.

Ao final do experimento, um questionário de feedback pedia ao participante para indicar a classe que ele achou mais difícil de compreender. As respostas a essa questão confirmaram os dados quantitativos: todos elegeram a classe Person2 como a classe mais difícil de compreender. O principal fator apontado pelos participantes como algo que dificultou o entendimento das classes foi o fato da classe analisada usar outras classes. Para eles, quanto mais dependente de outras classes mais difícil foi para entender a classe. É importante destacar aqui que os participantes não tiveram acesso ao código fonte das outras classes dos sistemas. Essa informação nos levou a coletar para cada uma das classes o valor de uma métrica de acoplamento. O objetivo foi saber até que ponto, as classes analisadas dependiam de outras classes do sistema. Usamos a bem conhecida

métrica Coupling Between Objects – CBO, definida por Chidamber & Kemerer [Chidamber & Kemerer 1994]. Os valores dessa métrica podem ser visualizados na Tabela 2.

Tabela 2. Valor do acoplamento das classes

Métrica	BibliotecaUI2	Person2	Locadora2	Contatos
CBO	3	5	3	1

A classe Person2 tem o maior grau de acoplamento (CBO = 5) dentre as quatro classes utilizadas: ela depende de cinco outras classes do sistema. Provavelmente tenha sido esse o motivo de sua compreensão ter sido considerada a mais difícil, tanto pelos resultados quantitativos, quanto pela opinião dos participantes. Esse resultado nos levou a planejar estudos para avaliar a relação entre acoplamento e esforço de compreensão como trabalhos futuros.

Diante dos resultados, discutimos as questões de pesquisa da seguinte forma:

***Questão 1*: A coesão de uma classe apresenta alguma influência sobre a compreensão do seu código?** Os resultados do estudo não foram suficientes para responder a essa questão. A classe que se mostrou mais difícil de entender (Person2) é a que tem coesão estrutural mais baixa. No entanto, parece que a dificuldade para compreender seu código deveu-se mais ao fato dela ser a classe que mais depende de outras classes do sistema. A classe mais fácil de entender (Contatos) foi a que apresentou coesão, tanto conceitual quanto estrutural, mais alta. No entanto, também é a classe com o menor grau de acoplamento.

***Questão 2*: Há diferenças no esforço de compreensão de classes com diferentes valores de coesão conceitual e coesão estrutural?** Os resultados do estudo também não foram suficientes para responder a essa questão. Se tomarmos com base as classes Person2 e Contatos, poderíamos dizer que coesão estrutural tem relação com compreensão. Porém, temos de novo o fator do alto acoplamento da classe Person2 influenciando. Por outro lado, as classes BibliotecaUI2 e Locadora2 tem: (i) aproximadamente o mesmo nível de esforço de compreensão, (ii) o mesmo grau de coesão conceitual (valores iguais de LCbC) e (iii) o mesmo valor de acoplamento. Como elas apresentam diferentes graus de coesão estrutural (LCOM5 = 0 para BibliotecaUI2 e LCOM5 = 0.6 para Locadora2) e demandam esforço similar para serem compreendidas, podemos dizer que, nesse caso, não houve relação entre coesão estrutural e esforço de compreensão.

5. Conclusão

O estudo preliminar desenvolvido teve objetivo de analisar em que nível o grau de coesão de classes está relacionado ao esforço para compreender o código fonte dessas classes. Os resultados obtidos não permitiram responder as questões de pesquisa, principalmente por não termos anulado um fator de confusão importante: o grau de acoplamento das classes. Um dos pontos positivos do estudo foi o fato de termos tido sucesso em medir o esforço de compreensão de classes.

Diante do contexto, pretendemos realizar outro experimento com um número maior de participantes e classes. Pretendemos tomar mais cuidado para minimizar os fatores de confusão, principalmente os valores de acoplamento das classes. Além disso,

esse estudo nos motivou também a realizar outro estudo para avaliar a relação entre o grau de acoplamento e o esforço de compreensão.

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 486662/2013-6)

Referências

Bois, B. D., Demeyer, S.; Verelst, J., Mens, T., Temmerman, M. (2006) "Does God Class Decomposition Affect Comprehensibility?". *International Conference on Software Engineering – IASTED*.

Briand, L. C., Daly, J. W. and Wust, J. K. (1998). "A Unified Framework for Cohesion Measurement in Object-Oriented Systems". *Empirical Software Engineering - An International Journal*, 3(1), pp. 65-117.

Chidamber, S.; Kemerer, C. (1994) "A Metric Suite for Object Oriented Design". *IEEE Transactions on Software Engineering*, 20(6), pp. 476-493.

Dunsmore, A.; Roper, M. (2000) "A Comparative Evaluation of Program Comprehension Measures". *EFoCS SEG Technical Reports*.

Feigenspan, J.; Apel, S.; Liebig, J.; Kastner, C. (2011) "Exploring Software Measures to Assess Program Comprehension". *IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Ferreira, K. A. M.; Bigonha, M. A. S.; Bigonha, R. S.; Almeida, H. C.; Neves, R. C. (2011) "Métrica de Coesão de Responsabilidade – A Utilidade de Métricas de Coesão na Identificação de Classes com Problemas Estruturais". *Simpósio Brasileiro de Qualidade de Software (SBQS)*. Curitiba - Paraná.

Henderson-Sellers, B., Constantine, L. and Graham, M. (1996) "Coupling and Cohesion (Towards a valid metrics suite for object-oriented analysis and design)". *Object Oriented Systems*, vol 3, pp. 143-158.

Liu, Y. et al. "Modeling class cohesion as mixtures of latent topics". (2009) Int'l Conf. on Software Maintenance (ICSM2009), September, Edmonton, p. 233–242.

Marcus, A. & Poshyvanyk, D. (2005) "The Conceptual Cohesion of Classes". Intl' *Conference on Software Maintenance (ICSM '05)*. Washington, DC, pp. 133-142.

Mondal, M.; Rahman, Md. S.; Saha, R. K.; Roy, C. K.; Krinke, J.; Schneider, K. A. (2011) "An Empirical Study of the Impacts of Clones in Software Maintenance". *IEEE 19th International Conference on Program Comprehension (ICPC)*.

Pfleeger, S.; Atlee, J. (2010) "Software Engineering: theory and practice", 4th ed, Prentice Hall.

Silva, B., Sant'Anna, C., Chavez, C. and Garcia, A. (2012) "Concern-Based Cohesion: Unveiling a Hidden Dimension of Cohesion Measurement". *IEEE International Conference on Program Comprehension (ICPC 2012)*, Passau, pp. 103-112.

Silva, B., Sant'Anna, C., Chavez, C. (2014) "An Empirical Study on How Developers Reason about Module Cohesion." *Int'l Conference on Modularity (Modularity 2014)*, Lugano, pp. 121-132.

Rugaber, S. (1995) "Program Comprehension". Working Report. Georgia Institute of Technology. May 1995.

Evaluation of Duplicated Code Detection Tools in Cross-Project Context

Johnatan A. de Oliveira[1], **Eduardo M. Fernandes**[1], **Eduardo Figueiredo**[1]

[1]Software Engineering Lab, Department of Computer Science,
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

{johnatan-si, eduardomorfernandes}@ufmg.br, figueiredo@dcc.ufmg.br

Abstract. *Two or more code segments are considered duplicated when there is a high rate of similarity among them or they are exactly the same. Aiming to detect duplicated code in single software projects, several tools have been proposed. However, in case of cross-project detection, there are few tools. There is little empirical knowledge about the efficacy of these tools to detect duplicated code across different projects. Therefore, our goal is to assess the efficacy of duplicated code detection tools for single projects in cross-project context. It was concluded that the evaluated tools has no sufficient efficacy in the detection of some types of duplicated code beyond exact copy-paste. As a result, this work proposes guidelines for future implementation of tools.*

1. Introduction

Duplicated code is a bad smell defined as replication of code segments through the source code of a software project. One of the problems derived from duplicated code is the high complexity of refactoring a system with high number of duplicated code. The reason is because a refactoring would demand changes in various parts of the code. The dissemination of errors through the system is another issue caused by duplicated code practices, because errors in a replicated segment are spread throughout the system [Fowler 1999].

In case of duplicated code detection in single software systems, there are many proposed tools [Juergens et al. 2009, Hotta et al. 2010] and comparative studies [Bellon et al. 2007]. However, in cross-project detection context, there is no empirical study. This type of detection may point to similar code segments that appears in different projects. Considering systems from a single business domain, it could support code reuse in development of projects [Bruntink et al. 2005], such as in a software product line (SPL).

This study aims to assess the efficacy of some duplicated code detection tools for single projects in cross-project context. For this purpose, it was conducted a literature review for identification of tools in order to assess their efficacy. As a result, this study pointed to a need of effective tools in the detection of more types of duplicated code than copy-paste. Furthermore, this work proposes some guidelines to support the development of duplicated code detection tools, based on demands detected through this study.

The remainder of this paper is organized as follows. Section 2 presents the background, describing the types of duplicated code and SPL, and related work that proposed comparison of duplicated code detection tools. Section 3 discusses the experimental

setup, including the research questions. Section 4 presents the results obtained through this study, including guidelines proposed for future implementation of detection tools. Finally, conclusions and future work are presented in Section 5.

2. Background and Related Work

According to [Fowler 1999], bad smells are symptoms of problems in source code. There are several bad smells defined in literature, but this study focuses on duplicated code, which has four types [Bellon et al. 2007]: *Type I* that is simple copy-paste, *Type II* that consists of copy-paste with simple variations in identifiers, literals, types, etc., *Type III* that is copy-past with variations in operations and code blocks, and *Type IV* that consists of code segments with the same computation, but different implementation.

A SPL is a development paradigm based on software platforms (common features among systems) and customizable features (accordingly to client needs). Systems of a SPL compose a software product family, and their commonalities are reused in the development of new products [Halmans and Pohl 2003]. Therefore, these systems can be useful to assess the efficacy of tools in the detection of duplicated code, and this study took advantage of this property to assess tools.

Previous studies assessed the efficacy of duplicated code detection tool. The work of [Bellon et al. 2007] compares six tools in order to identify their recall and precision, considering the detection techniques used by each tool and using C and systems developed in Java programming languages, they noted that the abstract syntax tree-based approaches tended to have higher precision, however the runtime is greater, token-based approaches had higher recall but were faster. In other direction, [Burd and Bailey 2002] focused on the evaluation of five duplicated code detection tools in order to identify the benefits of detection for preventative maintenance. In his work too have shown that token based techniques have a high recall but suffer from many false positives.

The present study, in turn, aims to evaluate a set with twenty duplicated code detection tools, in order to assess their efficacy in the context of cross-project detection, as well as to report the main features and issues of each tool. Furthermore, this study aims to conduct tests, for each tool, using as entry a set with twenty-one real software projects from an specific business domain. Its goal is to assess the efficacy of tools in the detection of the four types of duplicated code.

3. Experimental Setup

This section presents the experimental setup for this study, including: the strategy for selecting target systems to be used as input in tool evaluation, the selection of duplicated code detection tools, and the procedures for installing, running and evaluating each tool using different sets of system projects as input. Each step of this study is illustrated in Figure 1 and described in the following subsections. In order to perform this study, the following research questions were conceived concerning duplicated code detection tools:

- *RQ1: What duplicated code detection tools have been reported in literature?*
- *RQ2: Are the identified tools able to detect the four types of duplicated code in cross-project context?*
- *RQ3: Are the evaluated tools able to detect cross-project duplicated code?*

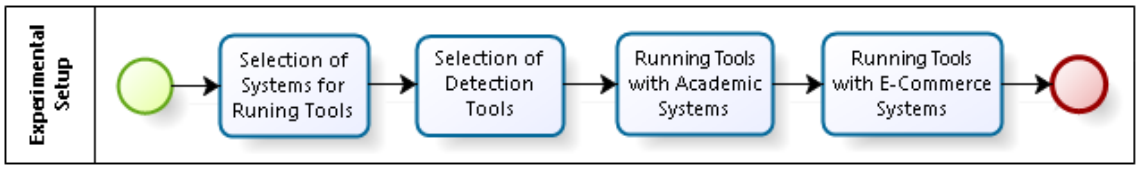


Figure 1. Steps of the conducted study.

3.1. Selection of Systems for Running Tools

In order to assess the efficacy of duplicated code detection tools through the running of each tool, two sets of software systems were selected: a systems academic (SPL) that was chosen because it allows us to evaluate whether the duplicated code tools are able to find reused code and e-commerce systems mined from GitHub[1]. The former was conceived to support the preliminary assessment of efficacy for each available tool, and the latter was conceived to be used in the assessment of tools filtered in the previous assessment.

The systems that compose the academic corpus were developed by graduate students through aspect-oriented programming with code reuse in an academic course between 2013 and 2014, at the Federal University of Minas Gerais (UFMG). These systems were implemented based on a platform system called SimulES, that consists of a software product line. They were chosen because of the conviction that exists duplicated code across all the systems, at least in respect to the used platform.

The e-commerce domain was chosen because of activities and the common functions across systems (e.g., purchases and payments). The process of system selection consisted of three steps: selection of 100 e-commerce bookmarked systems from GitHub, cleaning of selected systems (files other than .java files were removed), and discard of systems not implemented in Java (79 were discarded, remaining 21 systems).

3.2. Selection of Detection Tools

Aiming to select duplicated code detection tools to be evaluated in this study, we conducted an *ad hoc* review in order to identify the most cited tools in the literature. This literature review was based on the protocol for systematic literature reviews [Kitchenham et al. 2009], that suggests the following steps: planning (including the definition of a review protocol), conducting (including the data extraction) and reporting (in this case, we present a table with the collected data).

The literature review returned 19 tools described in Table 1. Columns in Table 1 indicate information about the tools: plug-in (PLG), developed programming language (PL), open-source or freeware (OSF), detection programming languages (DE), other detected bad smells (OTHER), is online (ON), documented (DO), graphical user interface (UI), detection technique (TE), and release year (YR) – N/A indicates that the information is not available in literature or website of the tools.

In order to reduce the amount of tools to be evaluated (given the infeasibility to evaluate all the found tools), the following inclusion criteria were defined for this study: the tool must be available online, it must to detect duplicated code in entire software projects, not in single source files only, it must have been cited in more than one paper,

[1]http://www.github.com/

it must be compatible with Java programming language, and it must be open source or freeware for non-commercial use.

In the selection of duplicated code detection tools, the tools CP-Miner [Li et al. 2004], Java CloneDR [Bellon et al. 2007] and Pattern Insight Clone Detection[2] were discarded because they are paid tools. Furthermore, the tools CCFinderX[3], CDSW [Murakami et al. 2013], Covet [Burd and Bailey 2002], Dup [Baker 1993], Duploc [Ducasse et al. 1999], Scorpio [Higo et al. 2013] and Simian[4] had to be discarded because they are not available online.

3.3. Running Tools with Academic Systems

This step consisted of installing and running tools in order to assess their efficacy in detection of duplicated code. These tools were installed in a Microsoft Windows 7 environment. DECKARD [Jiang et al. 2007] and DuDe [Wettel and Marinescu 2005] were discarded because it was not possible to install them. For this purpose, it was used the academic corpus as input, that is a SPL (in case, SimulES) and, therefore, the evaluated tools should be able to identify similar code in the SPL.

Considering that some tools were discarded in the previous step, only Checkstyle [Moha et al. 2010], Clone Digger [Bulychev and Minea 2008], CodePro Analytix[5], Condenser [Mealy et al. 2007], PMD [Juergens et al. 2009], SDD[6], and SonarQube [Campbell and Papapetrou 2013] were evaluated in this step. The process of running the selected tools with the academic corpus consists of SimulES been submitted to clone detection, whose results based the filtering of tools to be used in the next study step.

3.4. Running Tools with e-Commerce Systems

After the evaluation of tools in the previous step, it was verified that only PMD and Atomiq are able to detect at least Type I of cross-projects duplicated code. Then, they were evaluated using the e-commerce corpus, in order to assess their efficacy in cross-project duplicated code detection in real software projects. Both tools display the source directory of each file involved in the duplicated code occurrence, so it was possible to identify cross-project detection.

4. Evaluation and Discussion

This section discusses the results obtained through this study, focusing on answering each research question (Section 4.1). Furthermore, Section 4.2 presents some guidelines to support future implementation of duplicated code detection tools.

4.1. Answering the Research Questions

RQ1: What duplicated code detection tools have been reported in literature? This study identified some tools through an *ad hoc* literature review described in Section 3.2. These tools are presented in Table 1. Among the 20 studied tools, we identified that: 10 are

[2]http://patterninsight.com/products/clone-detection/
[3]http://www.ccfinder.net/ccfinderx.html
[4]http://www.harukizaemon.com/simian/
[5]https://developers.google.com/java-dev-tools/download-codepro
[6]http://sourceforge.net/projects/sddforeclipse/

Table 1. Tools for detection of duplicated code.

Tool	PLG	PL	OSF	DE	OTHER	ON	DO	UI	TE	YR
Atomiq	No	N/A	✓	C, C++, C#, Java, others	None	✓	×	✓	Token	2005
CCFinderX	No	C++	✓	C, C++, Java, C#, COBOL, others	None	×	✓	✓	Token	2005
CDSW	N/A	N/A	N/A	N/A	N/A	×	×	N/A	State-ment	2013
CheckStyle	Yes	Java	✓	Java	Large class, long method, long parameter list	✓	✓	✓	N/A	2001
Clone Digger	N/A	Python	✓	Java, Lua, Python	None	✓	✓	✓	Tree	2008
CodePro Analytix	Yes	Java	×	Java	None	✓	✓	✓	N/A	2001
Condenser	No	Jython	✓	Java, Python	None	✓	✓	×	N/A	2002
Covet	N/A	Java	✓	Java	None	×	×	N/A	Metric	1996
CP-Miner	N/A	Java	✓	Java	None	×	×	N/A	Token	2006
DECKARD	No	C	✓	Java	None	✓	✓	×	AST/ Tree	2007
DuDe	No	Java	✓	Java	None	✓	✓	×	N.F.	2010
Dup	N/A	C, Lex	N/A	C	None	×	×	N/A	Token	1995
Duploc	No	Small-talk	N/A	Inde-pendent	None	×	×	✓	Text	1999
Java CloneDR	N/A	N/A	×	C, C++, Java	None	✓	✓	✓	AST/ Tree	1998
Pattern In-sight Clone Detection	No	N/A	×	C, C++, Java, others	None	✓	✓	✓	Data Min-ing	N/A
PMD	Both	Java	✓	C, C++, C#, PHP, Java, others	God class, duplicated code, others	✓	✓	✓	N.F.	2002
Scorpio	N/A	Java	✓	Java	None	×	×	N/A	PDG/ Graph	2013
SDD	Yes	Java	✓	Java	None	✓	✓	✓	N/A	2005
Simian	Both	Java, .NET	×	C, C++, C#, Java, others	None	×	✓	✓	N/A	2003
SonarQube	No	Java	✓	C, C++, C#, Java, PHP, others	Various (manual metrics analysis)	✓	✓	✓	N/A	2008

compatible with Java, 12 are open source or freeware, 15 are able to detect only duplicated code, 11 have graphical user interface and 11 are available online.

RQ2: Are the identified tools able to detect the four types of duplicated code in cross-project context? As mentioned in Section 3.3, among the tools selected though the literature review, only PMD and Atomiq were able to identify at least one of the duplicated code types defined in literature (in case, only Type I).

RQ3: Are the evaluated tools able to detect cross-project duplicated code? It was verified that PMD and Atomiq were able to detect only Type I duplicated code among different systems. In this case, it was detected duplicated code among the systems that shared a single framework JadaSite[7]. Although the tools were able to detect cross-projects duplicated code in the e-commerce systems, it was difficult to identify the source origin of duplicated segments because the tools have no usability support for this kind of detection.

4.2. Guidelines for Development of Duplicate Detection Tools

Though the conducted study with a set of tools for detection of duplicated code, it was conceived five guidelines to support future implementation of tools for this purpose.

G1: To improve the tool usability, in order to minimize the required effort to use a tool and to understand its outputs. The evaluated tools presented some usability issues such as: difficulty to navigate between duplicated code occurrences (in general, results are showed in long lists without result cataloging), difficulty to identify the source file for each duplicated code (they display only directory path and not source project), lack of duplicated code highlighting, lack of click-to-open file for each duplicated code occurrence, and lack of feature for zooming results.

G2: To provide results analysis through statistical methods, graphical visualization or at least numerical indicator such as percentage of lines of duplicated code. It was identified that the evaluated tools do not show statistical numbers related to the duplicated code detection or amount of detected duplicated code for each type. These data could be useful, for instance, in result analysis and comparison of different detection tools.

G3: To combine different techniques (token, tree, etc.) for duplicated code detection, in order to improve the precision of detection results. It could be interesting to combine different techniques that can be more useful to detect an specific type of duplicated code, in order to increase the precision of the tool.

G4: To export result though different file formats, in order to support further analysis and integration with other tools. A feature to result export could be useful to provide an external validation of the result, as well as other processing intended by user.

G5: To provide the selection of projects from different directories to be submitted to duplicated code detection. A practical selection of projects from different data sources. This feature would avoid the cost of creating a single project with all the source code of projects to be analyzed.

4.3. Threats to Validity

The validity of the findings may have been affected by limitations such as: it was conducted an *ad hoc* review though, to minimize the threats, the review was inspired by

[7]https://github.com/IT-University-of-Copenhagen/JadaSite

systematic literature review protocol [Kitchenham et al. 2009]; this study focused only on academic tools, open source or free to use and compatible with Java, although it was collected a large set of tools; the conducted study used projects from e-commerce domain only; and lack of knowledge about the tools may have led to a inappropriate use of tools, though we used default settings.

5. Conclusion and Future Work

Duplicated code is a bad smell that can harm the software development, damaging the software maintenance. In this context, the duplicated code detection can be useful to improve the software quality. However, the identification of cross-projects duplicated code can point to relevant information such as the commonalities among from different projects and can be useful in the feature extraction to compose new software products such as in a SPL [Halmans and Pohl 2003].

Through this study, it was not possible to identify an efficient tool for cross-project duplicated code detection because the tools were able to find only Type I. Aiming to contribute for the community of duplicated code detection, guidelines were proposed to support future implementations of tools. Among the difficulties and challenges faced during the development of this study, we can cite: lack of tool documentation, inconsistency of documentation, online unavailability of tools, and usability issues of tools.

A suggestion for further work is the development a new cross-projects duplicated code detection tool following, as much as possible, the guidelines proposed in this study. These guidelines include statistical analysis for indication of duplicated code types proposed in the literature, and combination of different detection techniques, usability, result export, and features for selection of systems for duplicate code detection.

References

Baker, B. S. (1993). A Program for Identifying Duplicated Code. *Journal of Computing Science and Statistics*, pages 49–49.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):577–591.

Bruntink, M., van Deursen, A., van Engelen, R., and Tourwe, T. (2005). On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering (TSE)*, 31(10):804–818.

Bulychev, P. and Minea, M. (2008). Duplicate Code Detection Using Anti-unification. In *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, number 2.

Burd, E. and Bailey, J. (2002). Evaluating Clone Detection Tools for Use During Preventative Maintenance. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 36–43.

Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Ducasse, S., Rieger, M., and Demeyer, S. (1999). A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 109–118.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley.

Halmans, G. and Pohl, K. (2003). Communicating the Variability of a Software-Product Family to Customers. *Journal of Software and Systems Modeling (SoSyM)*, 2(1):15–36.

Higo, Y., Murakami, H., and Kusumoto, S. (2013). Revisiting Capability of PDG-based Clone Detection. Technical Report, Graduate School of Information Science and Technology, Osaka University.

Hotta, K., Sano, Y., Higo, Y., and Kusumoto, S. (2010). Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software. In *Proceedings of the joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105.

Juergens, E., Deissenboeck, F., and Hummel, B. (2009). CloneDetective – A Workbench for Clone Detection Research. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 603–606.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. (2009). Systematic Literature Reviews in Software Engineering – A Systematic Literature Review. *Journal of Information and Software Technology*, 51(1):7–15.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2004). CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, volume 4, pages 289–302.

Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. (2007). Improving Usability of Software Refactoring Tools. In *Proceedings of the 18th IEEE Australian Software Engineering Conference (ASWEC)*, pages 307–318.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36.

Murakami, H., Hotta, K., Higo, Y., Igaki, H., and Kusumoto, S. (2013). Gapped Code Clone Detection with Lightweight Source Code Analysis. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*, pages 93–102.

Wettel, R. and Marinescu, R. (2005). Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 8–pp.

Boas e Más Práticas no Desenvolvimento Web com MVC: Resultados de Um Questionário com Profissionais

Maurício F. Aniche[1], **Marco A. Gerosa**[1]

[1]Universidade de São Paulo (USP)
Departamento de Ciência da Computação

{aniche, gerosa}@ime.usp.br

Abstract. *Web applications are commonly built upon different technologies and architectures on both the server and client side. Among them, many applications rely on the MVC architectural pattern. As each layer is different from the other, understand their specific best practices is fundamental. In this study, we applied a questionnaire to identify and understand good and bad web software development practices according to professional developers. Based on 41 participants' answers in a questionnaire with 50 open and closed questions, we catalogued and ranked different best practices. We noticed that developers have a set of good practices that go beyond the existent MVC's architectural restrictions.*

Resumo. *Aplicações web são comumente construídas sobre diferentes tecnologias e arquiteturas, tanto no servidor e do lado do cliente. Dentre elas, muitas aplicações fazem uso do padrão arquitetural MVC e, uma camada é diferente da outra, entender as boas práticas específicas de cada camada é fundamental. Neste estudo, aplicamos um questionário para identificar e compreender boas e más práticas de desenvolvimento de software web na opinião de desenvolvedores profissionais. Baseado em 41 respostas dos participantes a um questionário com 50 perguntas abertas e fechadas, percebemos que os desenvolvedores possuem um conjunto de boas práticas que vão além das restrições arquiteturais existentes no MVC.*

1. Introdução

Não há mais a necessidade de se argumentar a importância do desenvolvimento para web. De acordo com o *Internet Live Stats* [ils 2015], em junho de 2015, o número de sites existentes em toda a rede mundial de computadores é perto de 1 bilhão. Desenvolver aplicações web é desafiador [Ginige and Murugesan 2001]. Afinal, elas são comumente compostas por diferentes tipos de tecnologias [Ward and Kroll 1999], tanto do lado do servidor, que faz todo o processamento e persistência de dados, quanto do lado do cliente, que apresenta e interage com o usuário.

Cada uma dessas tecnologias contém seu próprio conjunto de boas práticas. Para exemplificar, há diversas boas práticas consideradas pelo mercado e academia para aplicações Java, que fazem uso de Spring MVC como arcabouço MVC, Hibernate para persistência dos dados em um banco de dados relacional e HTML, JSP e JSTL para a camada de visualização. Em um projeto como esse, o desenvolvedor possivelmente pergunta à si

mesmo: *"Qual a melhor maneira de modelar essa classe de domínio?"*, ou *"Como escrever essa consulta SQL?"*, ou mesmo *"Essa regra pertence a essa classe controladora ou aquela classe de negócio?"*.

A busca por boas práticas é fundamental. No entanto, afirmar com certeza que uma prática é realmente eficaz é difícil. Afinal, boas e más práticas são naturalmente empíricas. Na própria definição de Booch [Booch 1998], em seu conhecido trabalho *"The Rational Unified Process: an introduction"* (conhecido também por RUP), "boas práticas" são abordagens que se mostraram válidas na indústria e que, quando combinadas, atacam as raízes de problemas de desenvolvimento de software; elas são boas práticas, não tanto porque elas conseguem quantificar seu valor, mas sim porque elas são comumente usadas na indústria em projetos de sucesso. Portanto, perguntamos:

> ***Q:*** *Quais são as boas e más práticas em desenvolvimento de software web, de acordo com profissionais da indústria?*

Neste trabalho, descrevemos uma análise exploratória das opiniões de desenvolvedores sobre as melhores práticas em desenvolvimento de aplicações Web que fazem uso do padrão arquitetural MVC [Krasner et al. 1988] – que é bastante comum nos arcabouços mais populares do mercado, como Spring MVC, Asp.Net MVC ou Rails.

2. Trabalhos Relacionados

Na indústria, a maior parte dos trabalhos preocupa-se com boas práticas em um nível mais abstrato, discutindo boas práticas de codificação, orientação a objetos e arquitetura. Na academia, os trabalhos focam na camada de visualização, com estudos sobre HTML, CSS e Javascript. Aqui, listamos alguns deles. No entanto, até onde conhecemos, não há estudos focados nas boas práticas específicas de cada camada.

A comunidade Java possui um grande catálogo de boas práticas. O mais popular deles, conhecido por *J2EE Patterns* [Alur et al. 2003], contém padrões que aparecem frequentemente em sistemas web. Fowler [Fowler 2002] também possui um catálogo de boas práticas, dentre os quais destacamos *Domain Model, Active Record, Data Transfer Object, Front Controller e Repository*.

A própria arquitetura MVC contém um conjunto de restrições para garantir a qualidade de código produzido. Todas regras de negócio são responsabilidade da camada de Modelo. Controladores não devem contê-las; eles devem apenas controlar o fluxo entre a camada de Modelo e Visualização. A camada de Visualização é responsável apenas por exibir e interagir com o usuário final.

Na academia, em sua dissertação de mestrado, Gharachorlu [Gharachorlu 2014] avaliou algumas más práticas em código CSS em projetos de código aberto. As más práticas foram levantadas após uma análise em alguns sites e blogs famosos na área. Segundo o autor, as mais frequentes são valores fixos no código, estilos que desfazem estilos anteriores e o uso de IDs nos seletores. Esse tipo de análise pode ser feita por ferramentas que fazem a análise automática das regras de um arquivo CSS [Mesbah and Mirshokraie 2012]. Badros *et al.* [Badros et al. 1999] chegaram inclusive a trabalhar em um conjunto de restrições, de maneira a diminuir os possíveis problemas que os desenvolvedores cometem. Já no trabalho de Nederlof *et al.* [Nederlof et al. 2014], os autores investigaram como

o código HTML de 4 mil aplicações web se comportaram. Eles encontraram diversos problemas, dentre eles, mensagens de erros e avisos da W3C, IDs não únicos na página, layout quebrado e problemas de acessibilidade.

3. Questionário

Conduzimos um questionário com objetivo de entender e descobrir quais são as boas e as más práticas do desenvolvimento web do ponto de vista dos profissionais. O questionário, entre abertas e fechadas, contém 51 perguntas. Todas as questões visam possíveis boas e más práticas e foram agrupadas em 3 seções, uma para cada camada do MVC. A estrutura de cada seção é similar: elas começam com perguntas fechadas e terminam com perguntas abertas. As perguntas fechadas são baseadas nas más práticas conhecidas pela indústria, nas quais os participantes devem avaliar sua severidade, em uma escala de 1 a 10. Todas elas foram derivadas de buscas por boas práticas em desenvolvimento web na comunidade, bem como na experiência dos autores deste trabalho. Nas perguntas abertas, os participantes podem escrever sobre suas boas e más práticas particulares, que não apareceram nas perguntas fechadas. Ao final, o participante também tem espaço para mencionar qualquer outra ideia que ele tenha.

Enviamos o questionário para a comunidade brasileira *.NET Architects*, que tem, até a data deste artigo, por volta de 2.000 usuários, e também postamos no perfil do Twitter dos autores, que tem 2.500 seguidores. O *tweet* foi passado adiante por mais 12 pessoas. Ao final, obtivemos 41 respostas de desenvolvedores de diferentes comunidades brasileiras. Todos os 41 participantes responderam às perguntas fechadas, e desses, 21 também responderam às perguntas abertas.

Analisamos o questionário em duas etapas. Para as perguntas fechadas, descrevemos os dados por meio de estatística descritiva. Já nas perguntas abertas, usamos processo de análise qualitativa e codificação, similar ao sugerido pela Teoria Fundamentada nos Dados [Strauss and Corbin 1997]. O protocolo de análise seguiu os seguintes passos: (i) leitura cada resposta, (ii) relacionar cada sentença a uma boa ou má prática a um código, (iii) categorizar, contabilizar e agrupar os códigos levantados. Ao final, obtivemos 61 códigos (boas e más práticas) diferentes[1].

4. Perfil dos Participantes

De maneira geral, os participantes possuem experiência relevante em desenvolvimento de software e de aplicações web. 38 dos 41 participantes tem mais de 3 anos de experiência em desenvolvimento de software, e 13 tem mais de 10 anos. Apenas um trabalha há menos de um ano como desenvolvedor de software.

Em relação à experiência em desenvolvimento web, os números, apesar de menores, também mostram que os participantes são experientes. A maioria deles desenvolve aplicações web entre 3 e 10 anos; apenas 8 participantes tem menos de 3 anos de experiência. Além disso, a maioria deles já colocou entre 3 e 10 aplicações em produção, e apenas 4 participantes só colocaram uma única. Em uma auto avaliação sobre conhecimentos em boas práticas, em uma escala Likert de 1 a 10, a média foi 7. No entanto, 8 deles deram nota menor ou igual a 4 para si mesmos.

[1]Todo o questionário, bem como as 41 respostas fechadas, 21 respostas abertas e os 61 códigos finais podem ser encontrados em http://www.aniche.com.br/cbsoft-vem-2015/.

Java é a linguagem mais popular entre os participantes; 25 deles a usam no dia a dia. Em segundo lugar, C#. Poucos participantes usam PHP e Ruby. Os arcabouços mais populares são JSF, e ASP.NET MVC, seguidos de Spring MVC e VRaptor. Acreditamos que a variedade de tecnologias beneficia nosso estudo, pois dessa forma evitamos práticas que são específicas de uma tecnologia.

5. Resultados

Ao longo do questionário, os participantes relataram diferentes boas e más práticas. Algumas delas, inclusive, foram relatadas por mais de um participante. Curiosamente, alguns participantes preferiram citar a má prática, enquanto outros preferiram destacar a boa.

Na Tabela 1, apresentamos as avaliações dadas por cada participante nas boas e más práticas das perguntas fechadas do questionário. Ela está em ordem de severidade (a mais severa primeiro). É possível notar que algumas más práticas já conhecidas na literatura são realmente problemáticas do ponto de vista do desenvolvedor, como a existência de regras de negócio fora da camada de modelo e a mistura de infraestrutura com código de domínio, enquanto outras são menos graves, como o caso da falta de injeção de dependência nos controladores, ou o uso de Javascript puro, sem arcabouços.

Tabela 1. Más práticas e seu nível de criticidade, de acordo com os participantes.

Camada	**Má prática**	**Mediana**	**8 a 10**	**6 a 8**	**<= 6**	**<= 2**
Modelo	Regras de negócio em DAOs	10.0	65%	15%	19%	2%
	Mistura de Regras de Negócio e Infraestrutura	9.0	50%	30%	19%	4%
	Complexidade	8.0	51%	34%	19%	7%
	Complexidade em DAOs	8.0	43%	29%	26%	2%
	Método ambíguo em DAOs	7.0	14%	48%	36%	4%
Controlador	Regras de negócio	9.0	53%	19%	26%	2%
	Complexidade	8.0	41%	29%	29%	4%
	Muitas rotas	6.0	17%	29%	53%	21%
	Falta de arcabouço de DI	5.0	19%	19%	6%	21%
Visualização	Regras de negócio	10.0	68%	17%	14%	2%
	CSS *inline*	8.0	26%	43%	29%	4%
	JavaScript arquivo único	8.0	46%	17%	36%	9%
	CSS em arquivo único	6.0	24%	19%	56%	31%
	Regras de visualização em scriptlets	5.0	17%	14%	65%	17%
	Longa Hierarquia em CSS	5.0	12%	24%	53%	17%
	JS sem *prototype*	5.0	7%	19%	7%	19%
	JS sem arcabouços	4.0	9%	12%	78%	29%

Nas subseções abaixo, apresentamos a análise do estudo qualitativo, feito em cima das 21 respostas abertas que obtivemos. Nelas, os participantes nos disseram suas particularidades nas suas boas e más práticas.

5.1. M - Modelo

A mistura de código de infraestrutura com código de domínio foi a má prática mais comentada entre os participantes. 10 deles explicitamente disseram acreditar que isso é um grave problema. Um dos participantes disse, traduzido do inglês: *"Em minha experiência, o grande problema são quando as classes domínio estão muito acopladas com infraestruturas externas, como ORMs. Isso faz com o que o modelo reflita os requisitos estabelecidos por essas dependências e, ao final, acabamos com um projeto de classes pobre. De vez em quando, isso é uma troca que os desenvolvedores fazem para ganhar produtividade e diminuir custos, mas normalmente isso causa muitos problemas no futuro."*. Um participante em particular disse que, do seu ponto de vista, uma boa prática e solução para o problema seria fazer com que as classes de modelo façam uso de um conjunto de abstrações que representem a infraestrutura.

Qualidade de código também apareceu como uma preocupação importante nas respostas do questionário. 3 participantes mencionaram o problema do alto acoplamento e baixa coesão; métodos longos e a falta de encapsulamento também foram mencionados. Em relação a boas práticas, três eles afiram que "o bom uso de POO" é fundamental; dois deles mencionaram o Princípio da Responsabilidade Única [Martin 2003]. O uso de abstrações, assim como imutabilidade em classes de domínio, injeção de dependências via construtor e *tiny types* [Hobbs 2007] também foram considerados boas práticas.

Em relação aos DAOs, classes que acessam dados em outras fontes, como bancos de dados, o problema mais reportado foi a existência de regras de negócios misturadas em seus códigos; 8 participantes comentaram. No entanto, opiniões diferentes também apareceram. Um deles disse que se uma regra de negócios específica melhorar a performance do sistema só por estar dentro do DAO, então isso seria válido. Dois outros desenvolvedores mencionaram que se a regra de negócios for simples, então ela também pode não ser um problema por estar dentro de um DAO. Outro disse que regras de validação de dados também são aceitáveis.

Além disso, o uso do padrão Repositório [Evans 2004], o uso de um arcabouço ou mini arcabouço para acessar os dados e a escrita de uma única consulta SQL por método também foram comentadas como boas práticas. Como más práticas, os participantes mencionaram o uso de carregamento preguiçoso (que é feito automaticamente por muitos arcabouços, como *Hibernate*) e o uso de uma "classe pai" que contém códigos genéricos para todos os DAOs do sistema.

Resumo. Desacoplar infraestrutura de regras de negócio, usar boas práticas de orientação a objetos nas classes de domínio. Ter apenas uma única consulta por método nos DAOs, usar o padrão Repositório. Evitar DAOs genéricos e carregamento preguiçoso (*lazy loading*).

5.2. C - Controlador

A má prática mais popular em controladores é também a existência de regras de negócios em seus códigos. 10 participantes escreveram sobre isso. Um deles foi bem claro sobre as responsabilidades de um controlador: *"Lógica de negócio deve ficar no modelo. Controller tem que ser o mais "burro" possível"*. No entanto, um participante mencionou que regras de negócio, como "checar se uma string é nula" ou "número é maior que X"

podem ser aceitáveis, assim como qualquer tratamento de exceção. Além disso, outro participante disse que regras para instanciar uma entidade são também válidas.

Muitos participantes mencionaram sobre controladores com mais responsabilidades do que deveriam. Dois deles afirmaram que muitas rotas em um único controlador pode ser um indicador disso. Um participante falou sobre a complexidade dos controladores, afirmando que muitas linhas de código são problemáticas. Interessantemente, outro participante disse não ligar para essa complexidade, contanto que elas não sejam regras de negócio: *"Eu sei que alguns podem discordar, mas eu tendo a colocar mais complexidade em meus controladores"*.

Resumo. Não ter regras de negócio dentro de controladores, com exceção de tratamento de exceções, validação e regras de criação de objetos. Evitar controladores com muitas responsabilidades e rotas.

5.3. V - Visualização

A existência de *scriptlets* (programação dentro de arquivos de visualização, como JSPs) com regras de negócio é a má prática mais popular na camada de visualização. Um participante mencionou que o uso de *scriptlets* são permitidos apenas para regras de visualização.

A falta de padrão e código macarrônico também apareceram como más práticas. Em termos de boas práticas, participantes mencionaram a criação de componentes de visualização para possibilitar reuso, validação do lado do cliente e um arcabouço para prover segurança de tipos entre as variáveis declaradas nesses arquivos.

Em relação a CSS, participantes deram muitas pequenas sugestões. No entanto, nenhuma delas foi dita por mais de um participante. Maus nomes, grandes hierarquias, CSS *inline*, unidades de medida absolutas e uso de ID e seletores complicados foram considerados más práticas. Como boas práticas, eles sugerem o uso de classes com nomes descritivos, evitar conflitos de especificidade de classes, não sobrescrever propriedades, usar ferramentas como LESS ou SASS, ter um CSS por JSP e separar CSS em componentes.

Em Javascript, a má prática mais popular foi o uso de variáveis globais. O uso de bibliotecas pesadas, como jQuery também foi mencionado como uma má ideia. Como boas práticas, participantes mencionaram o uso de um carregador de módulos, a separação entre Javascript e CSS e alta testabilidade. Ter um único código fonte com todo Javascript parece problemático. Participantes também acreditam que o não uso de protótipos, a maneira de simular orientação a objetos em códigos Javascript não é um problema.

Resumo. *Scriptlets* são permitidos apenas para regras de visualização. Ter um arquivo CSS por JSP, sem longas hierarquias e sem propriedades sobrescritas. Usar um carregador de módulos Javascript e pensar em sua testabilidade.

6. Discussão

Muitas das boas e más práticas levantadas pelos participantes já eram conhecidas. Afinal, várias delas dizem respeito às responsabilidades de cada camada do MVC. No entanto, acreditamos que os "casos especiais" mencionados pelos participantes são bastante valiosos. Por exemplo, controladores não podem conter regras de negócio, mas sim apenas

fluxo, o que já era sabido. No entanto, regras de validação, criação de objetos e tratamento de exceções são válidas. Além disso, eles devem ser pequenos e conter apenas poucas responsabilidades. Além disso, apesar de alguns arcabouços como ASP.NET MVC, VRaptor e Spring MVC sugerirem o uso de injeção de dependências, e outros, como Ruby on Rails e Django, não, os participantes não veem diferença de qualidade entre usar ou não.

Modelos devem conter regras de negócio, como também é dito no padrão MVC. No entanto, desenvolvedores sugerem forte uso de boas práticas de orientação a objetos e desacoplamento da infraestrutura. Já DAOs devem conter uma única consulta por método e fazer uso do padrão Repositório. DAOs devem evitar o uso de carregamento preguiçoso e a existência de regras de negócio.

Visualizações também devem evitar regras de negócio. *Scriptlets* são permitidos apenas para regras de visualização. CSS deve ser claros, conter nomes descritivos, hierarquias pequenas, e não devem se conectar com elementos pelos seus IDs. Javascript devem ser testáveis e fazer uso de um carregador de módulo. Além disso, ter um único CSS e Javascript por página é considerado uma boa prática.

7. Ameaças à Validade

Poucos participantes. Tivemos apenas 41 participantes, dos quais só 21 responderam às perguntas fechadas, o que pode ser considerado um número baixo para um estudo qualitativo. No entanto, acreditamos que a amostra selecionada aborda um perfil diversificado, e a análise qualitativa foi bastante rica. E, portanto, acreditamos que os achados são relevantes.

Localização. O questionário foi compartilhado apenas com a comunidade brasileira de desenvolvimento de software, e por isso podem haver algumas particularidades regionais. No entanto, como um desenvolvedor de software comumente lê conteúdo de diferentes programadores ao redor do mundo, acreditamos que o sentimento do desenvolvedor é também baseado nessas opiniões, diminuindo o possível problema da regionalização.

Peso das opiniões. Não demos pesos diferentes para opiniões que vieram de desenvolvedores mais experientes. Todas práticas mencionadas por qualquer participante apareceram neste estudo.

Anedotais. Não fizemos um experimento controlado para dizer se a prática é realmente boa ou ruim; elas refletem a opinião dos desenvolvedores. No entanto, como já discutido, boas práticas tendem a emergir de experiências anteriores desses desenvolvedores, e por isso, acreditamos elas venham de sua vivência.

8. Conclusão

Neste estudo, coletamos opiniões de 41 diferentes desenvolvedores sobre boas e más práticas em desenvolvimento web. Como pudemos ver, as boas práticas vão além das sugeridas pelo padrão arquitetural MVC. Cada camada tem o seu conjunto de boas e más práticas específicas.

Acreditamos que desenvolvedores web devam conhecer esse conjunto de boas e más práticas, pois elas podem ajudá-los a melhorar suas aplicações e evitar a criação de código de baixa qualidade. Apesar delas refletirem apenas o ponto de vista dos partici-

pantes, é importante para um desenvolvedor conhecer esses diferentes pontos de vista. Eles podem, junto com a equipe, decidir se as aplicarão ou não.

No entanto, ainda há trabalho a fazer. Perguntas que ainda temos em aberto: Quais más práticas são mais graves? Quais não são tão importantes? Como podemos detectar a presença dessas más práticas em aplicações web, de maneira automática?

Referências

(2015). Internet live stats. http://www.internetlivestats.com. Acessado em 14 de março de 2015.

Alur, D., Malks, D., Crupi, J., Booch, G., and Fowler, M. (2003). *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc.

Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. (1999). Constraint cascading style sheets for the web. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 73–82. ACM.

Booch, G. (1998). Software development best practices. *The Rational Unified Process: an introduction*, pages 3–16.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Gharachorlu, G. (2014). Code smells in cascading style sheets: an empirical study and a predictive model.

Ginige, A. and Murugesan, S. (2001). Web engineering: A methodology for developing scalable, maintainable web applications. *Cutter IT Journal*, 14(7):24–35.

Hobbs, D. (2007). Tiny types. http://darrenhobbs.com/2007/04/11/tiny-types/. Acessado em 14 de março de 2015.

Krasner, G. E., Pope, S. T., et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49.

Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

Mesbah, A. and Mirshokraie, S. (2012). Automated analysis of css rules to support style maintenance. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 408–418. IEEE.

Nederlof, A., Mesbah, A., and Deursen, A. v. (2014). Software engineering for the web: the state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 4–13. ACM.

Strauss, A. and Corbin, J. M. (1997). *Grounded theory in practice*. Sage.

Ward, S. and Kroll, P. (1999). Building web solutions with the rational unified process: Unifying the creative design process and the software engineering process. *Rational Software Corporation*.

Swarm Debugging: towards a shared debugging knowledge

Fabio Petrillo[1], **Guilherme Lacerda**[1,2], **Marcelo Pimenta**[1] **e Carla Freitas**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

[2]Faculdade de Informática - Centro Universitário Ritter dos Reis (Uniritter)

{fspetrillo,gslacerda,mpimenta,carla}@inf.ufrgs.br

Abstract. *Debugging is a tedious and time-consuming task since it is a methodical process of finding causes and reducing the number of errors. During debugging sessions, developers run a software project, traversing method invocations, setting breakpoints, stopping or restarting executions. In these sessions, developers explore some project areas and create knowledge about them. Unfortunately, when these sessions finish, this knowledge is lost, and developers cannot use it in other debugging sessions or sharing it with collaborators. In this paper, we present Swarm Debugging, a new approach for visualizing and sharing information obtained from debugging sessions. Swarm Debugging provides interactive and real-time visualizations, and several searching tools. We present a running prototype and show how our approach offers more useful support for many typical development tasks than a traditional debugger tool. Through usage scenarios, we demonstrate that our approach can aid developers to decrease the required time for deciding where to toggle a breakpoint and locate bug causes.*

1. Introduction

Developers usually spend over two thirds of their time investigating code – either testing or doing dynamic investigation using a debugger or reading, statically following methods' calls or using other source browsing tools [LaToza and Myers 2010]. Since debugging is a tedious and time-consuming task[Fleming et al. 2013], in the last 30 years, numerous approaches and techniques have been proposed to help software engineers in debugging. However, recent work showed that empirical evidence of the usefulness of many automated debugging techniques is limited, and they are rarely used in practice[Parnin and Orso 2011]. Thus, if developers spend a lot of time debugging code, why should this human effort be lost? Could we collect debugging session information to use in the future, creating visualizations and seaching tools about this information? Why is a breakpoint toggled? What is its purpose?

To address these debugging issues, we claim that *collecting and sharing debugging session information can provide data to create new visualizations and searching tools to support programmers tasks, decreasing the time for developers deciding where to toggle the breakpoints as well as improving project comprehension*. To support this statement, we propose a new approach named Swarm Debugging. Swarm Debugging is a technique to collect and share debugging information, and to provide visualizations and searching tools for supporting the debugging process.

The remainder of the paper is structured as follows. Section 2 describes Swarm debugging, its structure, visualizations and searching tools. Section 3 discusses its results

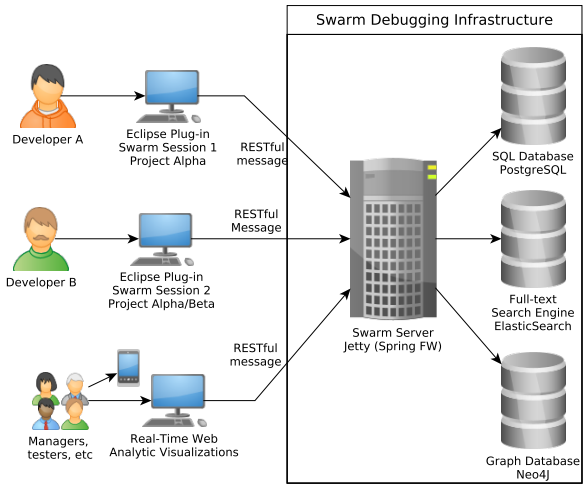


Figure 1. The Swarm Debugging Infrastructure

and main contributions compared to related work, and in Section 5 we draw some final comments and present future work.

2. The swarm debugging approach

Swarm debugging (SD) is an approach inspired by three challenges in software engineering: the time and effort spent during debugging sessions, the difficulties of deciding where to set a breakpoint, and how collective intelligence can be used to improve software development. Firstly, developers spend long periods in debugging sessions, finding bugs or understanding a software project [LaToza and Myers 2010]. During these sessions, while using traditional debugging tools, they collect a lot of information and create a mental model of the project [Murphy et al. 2006]. Unfortunately, several studies have shown that developers quickly forget details when they explore different points within the source code, loosing parts of the mental model [Tiarks and Röhm 2013]. The second challenge is finding suitable breakpoints when working with the debugger [Tiarks and Röhm 2013]. Developers have to find the adequate breakpoints in order to reproduce the data and control flow of the program. Tiarks and Röohms [Tiarks and Röhm 2013] observed that developers encountered problems to find the appropriate breakpoints, because deciding where to toggle a breakpoint is an “extremely difficult" task. They found that programmers applied a strategy: they set a lot of breakpoints in the beginning, then they start the debugger, and after that, they remove the breakpoints that turned out to be irrelevant. They noticed some programmers that wrote notes on a piece of paper or used an external editor as a temporary memory. Moreover, the study suggested that this strategy causes significant overhead. When a developer does not know exactly where an error is located, he or she chooses a breakpoint as a starting point or a line of code near the point that is an hypothesis for this error. Thus, this point is usually an strategic area in the software, and this knowledge is lost in the current debugging tools. The third challenge is how collective intelligence can be used in software development. Software development is a cooperative effort [Fuggetta 2000], and Storey et al. [Storey et al. 2014] claim that collective intelligence is an open field for new software development tools.

Swarm Debugging is based on three works: 1) the declarative and visual debugging environment for Eclipse called JIVE[Czyz and Jayaraman 2007], 2) a novel user in-

terface to support feature location named In Situ Impact Insight (I3) [Beck et al. 2015], and 3) the Kononenko et al.'s experience with ElasticSearch for performing modern mining software repositories research [Kononenko et al. 2014]. Firstly, JIVE is an Eclipse plug-in that is based on the Java Platform Debugging Architecture (JPDA), which allows the analysis of a Java program execution. JIVE requests notification of certain runtime events, including class preparation, step, variable modification, method entry and exit, thread start and end. Our approach also uses JPDA to collect debug information. Secondly, I3 [Beck et al. 2015] introduces a novel user interface, which allows developers to retrieve details about the textual similarity of a source code and to highlight code in the editor, as well as augment it with *in situ* visualizations. These visualizations also provide a starting point for following relations to textually similar or co-changed methods. Swarm Debugging uses textual similarity in its searching tools on software projects repositories. Finally, as Kononenko et al. [Kononenko et al. 2014], we use ElasticSearch to create a powerful search mechanism for the collected data during the debugging sessions, providing a shared information retrieval system for software developers. So, as debugging is a foraging process [Fleming et al. 2013], the Swarm Debugging main idea is to collect information during the sessions, storing breakpoints, reachable paths and debugging behaviors. Finally, these data are transformed into knowledge by visualizations and searching tools, creating a collective intelligence about software projects. In the next sections, we describe how Swarm Debugging implements this idea.

2.1. Architecture and infrastructure

Integrated Development Environments (IDEs) are widely used tools to develop software [Minelli et al. 2014]. We chose Eclipse[1] to implement our prototype. The Java Development Tools (JDT) is an important component of Eclipse, which includes debugging support in the JDT Debug module. Eclipse uses JPDA in order to provide traditional debugging capabilities such as setting of breakpoints, stepping through execution, or examining variables in the call stack. The Swarm Debugging prototype (Figure 1) is an Eclipse Plug-in, which captures Java Platform Debugging Architecture (JDPA) events in a debugging session.

When a developer creates a session, the SD starts two listeners, and it uses RESTful messages to communicate with a Swarm Server Infrastructure. These messages are received by a Swarm Server instance that storages them in three specialized persistent components: a SQL database (PostgreSQL), a full-text search engine (ElasticSearch) and a graph database (Neo4J). So, the Swarm Server provides an infrastructure for sharing information between developers, searching tools, and, as a web container (Jetty), for creating our visualizations (see section 2.2). During the debugging session, for each breakpoint or *Step Into* event, stack trace items are analysed by a tracer, extracting method invocations. For each pair of invocation, the tracer obtains an invoking and invoked method, creating an invocation entry.

2.2. Visualization techniques and searching tools

Swarm Debugging (SD) implemented the visualizations using the CytoscapeJS [Saito et al. 2012], a JavaScript Graph API framework. As a web application, the SD

[1]https://www.eclipse.org/

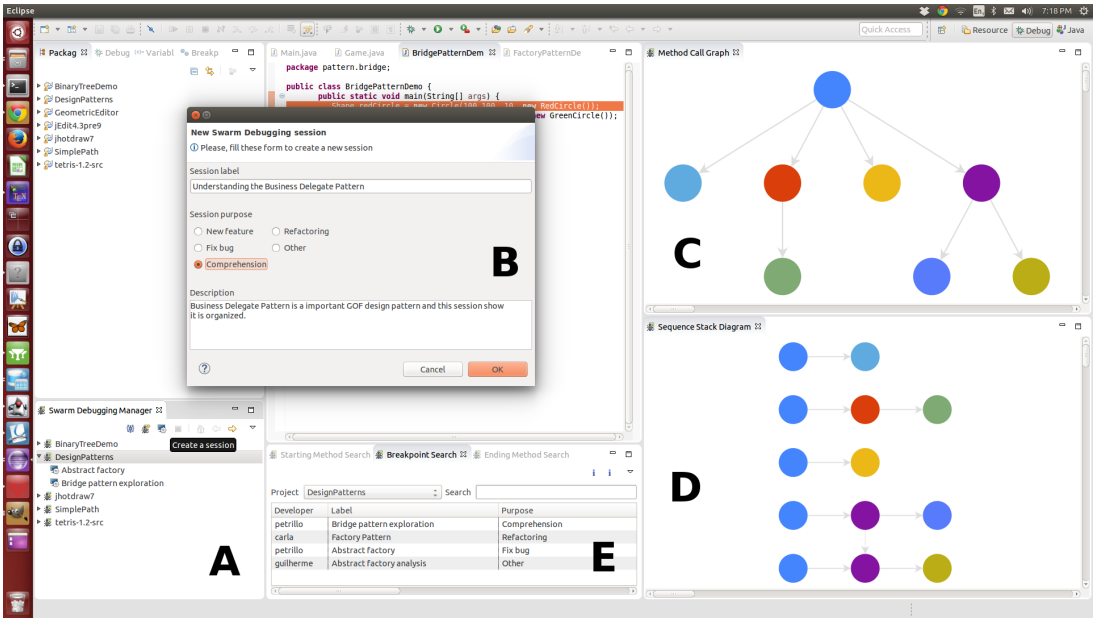


Figure 2. Swarm Debugging View

visualizations can be integrated into the Eclipse view as a SWT Browser Widget, or accessed through a traditional browser as Mozilla Firefox or Google Chrome, or web browsers for tablets and smartphones. Using these technologies, SD provides several visualizations about collected information: 1) Sequence stack diagram; 2) Dynamic method call graph; and 3) Swarm dashboard. Figure 2 shows the SD interface.

2.2.1. Sequence stack diagram

The sequence stack diagram is a novel way to represent a sequence of methods invocations (Figure 2-D). We use circles to represent methods and arrows to represent invocations. Each line is a complete stack trace, without returns. The first node is a Starting point (non-invoked method), and the last node is an Ending point (non-invoking method). If the invocation chain continues to a non-starting point method, a new line is created, the current stack is repeated, and a dotted arrow is used to represent a return for this node. In addition, developers can directly go to a method in the Eclipse Editor by double-clicking on its node in the diagram.

2.2.2. Dynamic method call graph

The dynamic method call graph is based on directed call graphs [Grove et al. 1997] (Figure 2-C) for explicitly modeling the hierarchical relationship induced by invocated methods. This visualization uses circles to represent methods (nodes) and oriented arrows to express invocations (edges). Each session generates a call graph and all invocations collected during the session are shown in this visualization. The starting points (non-invoked methods) are represented at the top of a tree, and the adjacent nodes represent the invocation sequence. In addition, the developer can navigate along the sequence of invocation

methods by pressing the F9 key (forward) or the F10 key (backwards). Finally, developers can go directly to a method in the Eclipse Editor by double-clicking on its node in the diagram.

2.2.3. Swarm dashboard

ElasticSearch technology brings us a powerful dashboard tool named Kibana. Using Kibana, we created an online panel to display project information. Several charts can be built to show the number of invocations, breakpoints or events by developers; the number of breakpoints by project and purpose; number of events by minute, etc.

Swarm Debugging also provides several searching tools for developers. To allow finding suitable breakpoints [Fleming et al. 2013] when working with the debugger, we developed a search tool. For each breakpoint toggle, SD captures the type and line of code where the breakpoint was toggled, storing this in the SD infrastructure databases. This way, all developers can share their breakpoints. The breakpoint search tool (Figure 2-E) combines *fuzzy*, *match* and *wildcard* ElasticSearch query strategies. Results are displayed in the search view table, which allows an easy selection. Finally, developers can open a type directly in Eclipse Editor by double-clicking on a selected breakpoint. The last search tool is a full-text search into the project source code. Eclipse IDE makes available a search tool, but it doesn't have full-text search features such as those provided by ElasticSearch. So, our source code search is a complementary tool for developers.

2.3. Swarm Debugging usage

This section describes the steps for using our approach. Firstly, using the Eclipse IDE, developers open a view "Swarm Manager", and establish a connection with the Swarm server (Figure 2-A). If the target project is not in the Manager, they associate a project from their workspaces to the Swarm Manager. Secondly, they create a Swarm session (Figure 2-A), informing a label, debugging purpose and a description for the new session (Figure 2-B). Automatically, the Eclipse Perspective is switched to *Debug*, and the visualizations *Method Call Graph* and *Sequence Stack Diagram* are shown (Figures 2-C and 2-D).

Then, with a session established, developers may need to toggle some breakpoints. For supporting this task, developers use breakpoint, starting point, or source search tools to find the correct location for their goal (Figure 2-E). Swarm Debugging captures and stores all breakpoints toggled by the developers. After adding all breakpoints, they start a conventional debugging execution, stopping in the first reached breakpoint. For each *Step Into* or *Breakpoint* event, SD DebugTracer captures the event and stores its method call, producing invocations entry for each pair invoking/invoked method. Automatically, the method call graph and sequence stack diagram visualizations are updated by the views' mechanisms.

The process continues until the execution is finished, when the Swarm Debugging session is completed. Finally, all collected information is made available for developers through visualization of the resulting graphs or breakpoints.

3. Discussion

During the SD implementation, we analyzed many usage scenarios, and we can summarize some practical observations. The breakpoint search tool is powerful and useful. After recording many breakpoints in the database, new debugging sessions were started much quickly because we found the significant breakpoints easily. In certain situations, although some breakpoints were not the exact points that we wanted, they were good points to start. So, the breakpoint search tool decreases the time spent to toggle a breakpoint.

Swarm sessions showed an important feature to divide the software complexity problem. In traditional approaches, as the JIVE approach, all data are used for visualizations and search. In addition, when the session finishes all data are lost. On the other hand, Swarm Debugging saves session information, and developers can divide massive projects in multiple diagrams, having a fine control to explore relevant or important portions of their software project. This feature improves the overall software project comprehension.

Nevertheless, it would be possible to argue that debugging information in different parts of the software would not be interesting to share. However, we believe that in certain situations, a developer can take advantage of his colleague's knowledge who has explored an unknown area before him. The problem to be solved might not be the same, but may be in the same region of the project, and knowing where some breakpoints were placed before (by himself or by other developers) could accelerate a decision of where to put another ones. However, since this is really a new proposal, for sure we need to evaluate it, and this is immediate future work.

One of the most important innovations in our proposal is that Swarm Debugging creates diagrams during a debugging session, and makes specific graphs for an execution. Furthermore, if we carefully observe a diagram, the redundancy for each stack line allows the developer to use zooming or panning to visualize exactly a sequence needed for understanding the software. We compared the sequence stack diagram with the UML sequence diagram, and our proposal is better for large sequences (we used JIVE to do this comparison). However, of course, we need to evaluate this experimentally to validate the hypothesis of usefulness of our approach.

4. Related work

In recent years, researchers have worked on improving debugger tools to address developers' issues. Our work is related to several aspects of software development, including debugging techniques, visual debugging, databases, and collective intelligence.

Debugging Canvas [DeLine et al. 2012] is a tool where developers debug their codes as a collection of code bubbles, annotated with call paths and variable values, on a two-dimensional pan-and-zoom surface. SD uses a similar idea, but it integrates source code and visualization in two distinct views. Estler et al. [Estler et al. 2013] discussed the *Collaborative Debugging* describing CDB, a debugging technique and integrated tool designed to support effective collaboration among developers during shared debugging sessions. Their study suggests that debugging collaboration features are perceived as important for developers, and can improve the experience in collaborative context. This result founded our approach, but differently of CDB (a synchronous debugging tool), SD is an asynchronous session debugging tool. Consequently, SD does not have several

collaboration issues. Minelli et al. [Minelli et al. 2014] presented a visual approach for representing development sessions based on the finest-grained UI events. The authors have collected, visualized, and analysed several development sessions for reporting their findings. SD is based on capturing UI events, but related to breakpoints.

5. Conclusions

In this paper, we have presented a new approach called Swarm Debugging aimed at visualizing and sharing information obtained from debugging sessions, and we have presented a novel tool as an Eclipse plugin. Our approach uses the developers workforce to create a context-ware visualization, automatically capturing and sharing knowledge with its context by observing developers' interactions that were discarded in traditional debugging tools. It allows programmers to find quickly breakpoints, starting points and *share their debugging experiences* about projects. By focusing context-aware sessions, each swarm debugging session captured only the **intentional exploration path**, focusing visualizations and searches on developer issues.

Our approach provides several contributions. First, a technique and model to collect, store and share debugging session information, contextualizing breakpoints and events during these sessions. Second, a tool for visualizing context-aware debugging sessions using call graphs and a map where developers can check areas that were visited by collaborators, and to know who already explored a project. Using web technologies, we created real-time and interactive visualizations, providing an automatic memory for developer explorations. Third, a tool for searching starting points and breakpoints for software projects based on shared session information collected by developers. Moreover, dividing software exploration by sessions and its call graph are easy to understand because just intentional visited areas are shown in the graph. So, Swarm Debugging is used to support developers in their everyday work and on how software comprehension features can be integrated into the workflows depending on the task. With swarm debugging developers can follow the execution and see only the points that are relevant to programmers. In contrast, when using traditional visual debugging, one must see whole executions and manipulate plenty of irrelevant paths.

Despite the promising results we obtained, this paper is a preliminary work towards new ways to share and visualize information about debugging sessions. As future work, first, we plan to extend our catalog of visualizations and enlarge our dataset of debugging sessions. Second, we plan to integrate our tool with a bug tracking system, improving the breakpoint search, associating issues track information with breakpoints. Finally, we will perform new empirical studies to test our approach.

Acknowledgments This work has been partially supported by CNPq.

References

Beck, F., Dit, B., Velasco-madden, J., Weiskopf, D., and Poshyvanyk, D. (2015). Rethinking User Interfaces for Feature Location. In *23rd IEEE International Conference on Program Comprehension*, Florence. IEEE Comput. Soc.

Czyz, J. K. and Jayaraman, B. (2007). Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange - eclipse '07*, pages 31–35.

DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., and Reiss, S. P. (2012). Debugger Canvas: Industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073. IEEE.

Estler, H. C., Nordio, M., Furia, C. a., and Meyer, B. (2013). Collaborative debugging. *Proceedings - IEEE 8th International Conference on Global Software Engineering, ICGSE 2013*, pages 110–119.

Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. (2013). An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology*, 22(2):1–41.

Fuggetta, A. (2000). Software process: a roadmap. volume 97, pages 25–34.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call graph construction in object-oriented languages. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, pages 108–124.

Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. (2014). Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 328–331, Hyderabad, India. ACM.

LaToza, T. D. and Myers, B. a. (2010). Developers ask reachability questions. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:185–194.

Minelli, R., Mocci, A., Lanza, M., and Baracchi, L. (2014). Visualizing Developer Interactions. In *2014 Second IEEE Working Conference on Software Visualization*, pages 147–156.

Murphy, G., Kersten, M., and Findlater, L. (2006). How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4).

Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis ISSTA 11*, page 199.

Saito, R., Smoot, M. E., Ono, K., Ruscheinski, J., Wang, P.-l., Lotia, S., Pico, A. R., Bader, G. D., and Ideker, T. (2012). A travel guide to Cytoscape plugins. *Nature methods*, 9(11):1069–76.

Storey, M.-a., Singer, L., Cleary, B., Filho, F. F., and Zagalsky, A. (2014). The (R)Evolution of Social Media in Software Engineering. *FOSE*.

Tiarks, R. and Röhm, T. (2013). Challenges in Program Comprehension. *Softwaretechnik-Trends*, 32(2):19–20.

JSCity – Visualização de Sistemas JavaScript em 3D

Marcos Viana, Estevão Moraes, Guilherme Barbosa,
André Hora, Marco Tulio Valente
{longuinho,estevaoma,barbosa,hora,mtov}@dcc.ufmg.br

[1]Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – Brasil

Abstract. *JavaScript is one of the most used languages on the web. A wide range of frameworks and libraries widely adopted on the market make use of Javascript. In order to support the development and maintenance of such systems, source code visual representations can be used for restructuring, refactoring and understanding JavaScript software. Although there are tools that generate visual representations of code in other languages such as CodeCity for Java, no similar tool is available for JavaScript applications. This paper presents JSCity, a tool for the interactive visualization of JavaScript systems in 3D, using a city metaphor. For this work, we analyzed 40 popular open source systems written in JavaScript hosted in GitHub.*

Resumo. *JavaScript é uma das linguagens mais utilizadas da web. Muitos frameworks e bibliotecas adotados no mercado fazem uso de JavaScript. Para dar suporte ao desenvolvimento e manutenção desses sistemas, representações visuais podem ser utilizadas na reestruturação, refatoração e entendimento de código. Embora existam ferramentas que gerem representações visuais de código em outras linguagens, como proposta pelo sistema CodeCity, nenhuma ferramenta realiza essas representações para sistemas em JavaScript. Esse artigo apresenta JSCity, uma ferramenta para a visualização de sistemas JavaScript em 3D usando a metáfora de uma cidade. Para esse trabalho, foram analisados 40 sistemas populares escritos em JavaScript, que estão hospedados no GitHub.*

1. Introdução

JavaScript é uma linguagem de programação cada vez mais popular, destacando-se cada vez mais no mercado. De acordo com o site `http://githut.info`, JavaScript é a linguagem mais popular do GitHub. Sua principal característica é a dinamicidade, pois executa comandos em um navegador sem recarregamento da página ou interpretação de servidor [ECMA International 2011]. A linguagem foi inicialmente concebida em meados da década de 1990 com o objetivo de estender páginas web com pequenos trechos de código executáveis. Desde então, sua popularidade e relevância só tem crescido [Kienle 2010] [Nederlof et al. 2014]. A linguagem atualmente é utilizada inclusive para implementar clientes de e-mail, aplicações de escritório, IDEs, dentre outros, que podem atingir milhares de linhas de código [Richards et al. 2010]. Junto com a sua crescente popularidade, o tamanho e a complexidade de sistemas JavaScript também está em constante ascensão.

Por outro lado, a compreensão de sistemas, mesmo que com baixa complexidade, é uma tarefa árdua e cara. Estima-se que a manutenção do software represente 90% dos

custos totais de um sistema [Aparecido et al. 2011] e que grande parte do tempo seja dedicado ao entendimento do software [Guha et al. 2010]. Nesse contexto, a visualização de software é uma técnica utilizada para ajudar os desenvolvedores. Com o auxílio de tecnologias de visualização gráfica, ferramentas de apoio podem ser desenvolvidas para representar diversos aspectos de um sistema, principalmente sua estrutura, comportamento e evolução [Stasko et al. 1997].

Nesse artigo, apresenta-se JSCity, uma adaptação da metáfora de cidades para a linguagem JavaScript. Através dessa metáfora, é possível representar funções e aninhamento de funções, além de identificar arquivos que agrupam essas funções. JSCity é uma ferramenta de código aberto que oferece aos desenvolvedores uma forma intuitiva de representar, modelar e visualizar grandes quantidades de dados de desenvolvimento por meio de uma cidade 3D, como originalmente proposto por Wettel e Lanza [Wettel et al. 2011] para sistemas Java. Essa visualização facilita o entendimento da organização do código de uma aplicação e oferece uma analogia visual para que equipes de desenvolvimento possam se comunicar de forma mais eficiente. A ferramenta foi utilizada para criar as representações de 40 sistemas populares hospedados no GitHub.

Desse modo, as principais contribuições desse trabalho são:

1. Adaptação da metáfora de cidade para a linguagem JavaScript, considerando suas principais estruturas e os usos mais comuns da linguagem;
2. Análise de JavaScript, visualização de funções, arquivos e diretórios;
3. Disponibilização de uma solução computacional para visualização das cidades em 3D facilmente acessível através de uma página web, facilitando compartilhamento entre desenvolvedores.

O artigo está organizado como descrito a seguir. Na Seção 2, descreve-se a metáfora da cidade para representação de código. A Seção 3 apresenta a ferramenta JSCity. Em seguida, na Seção 4, são descritos casos de uso da ferramenta. Finalmente, apresentam-se os trabalhos relacionados na Seção 5 e as conclusões na Seção 6.

2. A Metáfora da Cidade em JavaScript

JavaScript é uma linguagem de programação dinâmica, sendo classificada como uma linguagem de script, baseada em protótipos, com tipagem dinâmica e funções de primeira classe [ECMA International 2011]. Assim, JavaScript é multi-paradigma, possibilitando programação orientada por objetos, imperativa e funcional. Apesar da possibilidade de representar classes em JavaScript por meio de protótipos [Silva et al. 2015], a metáfora proposta não representa classes diretamente e sim funções, que são as principais estruturas utilizadas em sistemas JavaScript.

O uso da metáfora de cidade para representação do código foi inspirado na ferramenta CodeCity [Wettel et al. 2011]. Uma cidade é uma forma intuitiva de representação, uma vez que está inserida em nossa vida cotidiana. Assim, propõe-se uma metáfora de cidade adaptada para JavaScript conforme apresentado na Figura 1. Nessa representação, distritos são diretórios, subdistritos são arquivos e prédios são funções.

Distritos e subdistritos são representados pelas cores amarelo e vermelho, respectivamente. Os prédios representam as funções, que podem ser funções anônimas (cor verde) ou funções nomeadas (cor azul). A altura dos prédios é representada pelo *número*

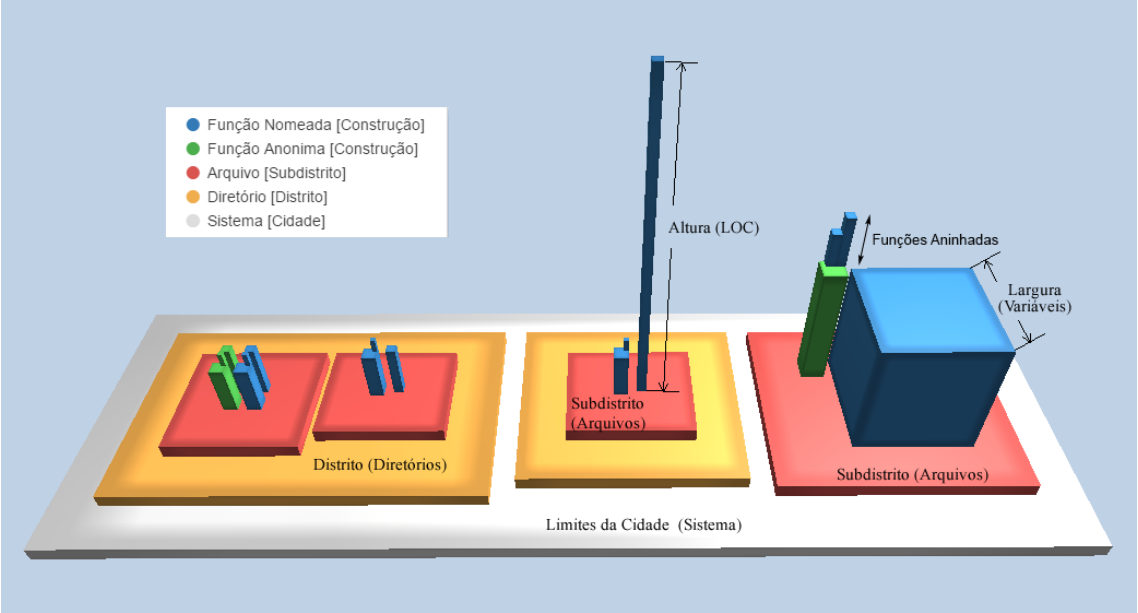


Figura 1: Princípios da metáfora da cidade

de linhas de código da função (LOC) e a largura é representada pelo *número de variáveis da função*.

Uma prática muito comum em JavaScript é declarar uma função dentro de outra, recurso esse denominado de funções aninhadas. Para esses casos, foi adaptado a metáfora da cidade para exibir um prédio sobre o outro. A altura total do prédio é o somatório das linhas de todas as funções aninhadas. A largura da função pai é o somatório das suas variáveis com as variáveis das suas funções filhas, garantindo a construção de um prédio mais largo na base e pequenos prédios acima deste.

A utilização de funções aninhadas é muito comum em JavaScript. Essa propriedade favorece a criação de estilos de programação, como por exemplo, criar uma estrutura que represente uma classe, declarar uma função e dentro dessas outras *n* funções e chamá-las da mesma forma que se chama um método em orientação a objetos. Dessa forma, entende-se que representar esse comportamento não apenas gera uma visualização mais intuitiva, como também possibilita diferenciar alguns padrões no código e possivelmente a distribuição arquitetural do projeto.

3. JSCity

A metáfora proposta foi implementada na ferramenta JSCity[1] para analisar sistemas desenvolvidos em JavaScript através de visualizações interativas de código em 3D. A construção da ferramenta foi realizada utilizando a própria linguagem JavaScript, o framework *Esprima* [Hidayat 2012], cujo papel é gerar uma Árvore Sintática Abstrata (AST) e o framework *ThreeJS* [2], cujo papel é desenhar toda a cidade. Como ilustrado na Figura 2, a ferramenta funciona em cinco passos:

[1] `https://github.com/ASERG-UFMG/JSCity/wiki/JSCITY`
[2] http://threejs.org

1. Execução de um script Node.js para analisar o código com o *Esprima* e gerar uma Árvore de Sintaxe Abstrata (AST) no formato JSON;
2. Interpretação da AST e persistência dos dados relativos às principais estruturas de JavaScript para representação da cidade. No final dessa etapa, os dados necessários para o desenho da cidade já estão gravadas no banco de dados.
3. Usuário escolhe repositório para visualização da cidade em uma página web;
4. Leitura da base de dados para desenho da cidade;
5. Desenho da cidade utilizando o framework *ThreeJS*.

O framework *ThreeJS* oferece recursos gráficos que possibilitam a representação de cenas em 3D. Dentre os recursos oferecidos, podemos enumerar a criação de cenas, geometrias, animações e perspectivas sem o recarregamento da página. Esses recursos foram utilizados pela ferramenta para desenhar a cidade e possibilitar a navegação pelos elementos da cidade, realizar operações de *zoom*, mudar o ângulo da câmera e posicionar o *cursor* no prédio para exibir os dados da função.

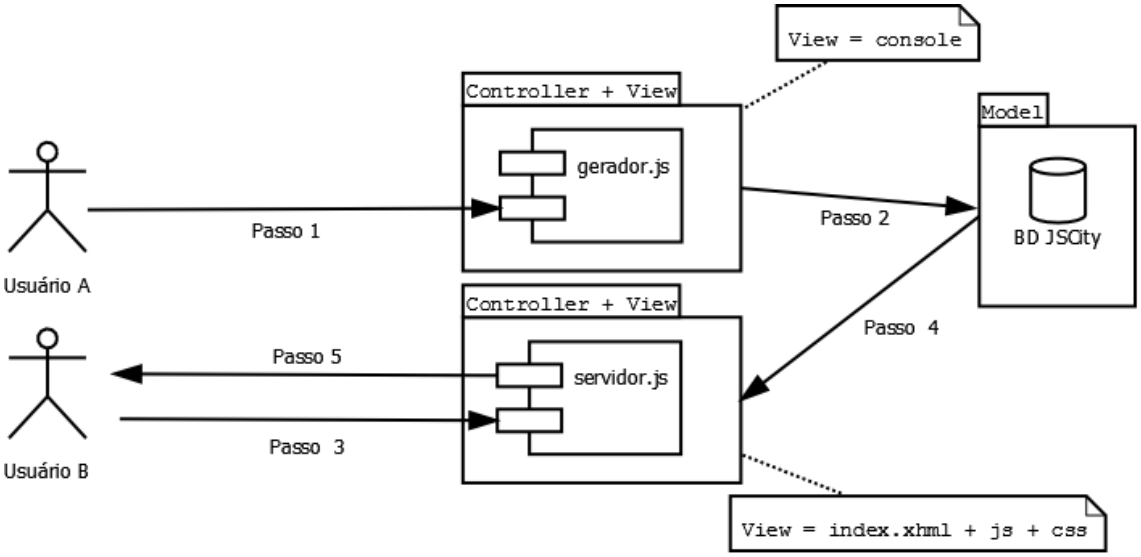
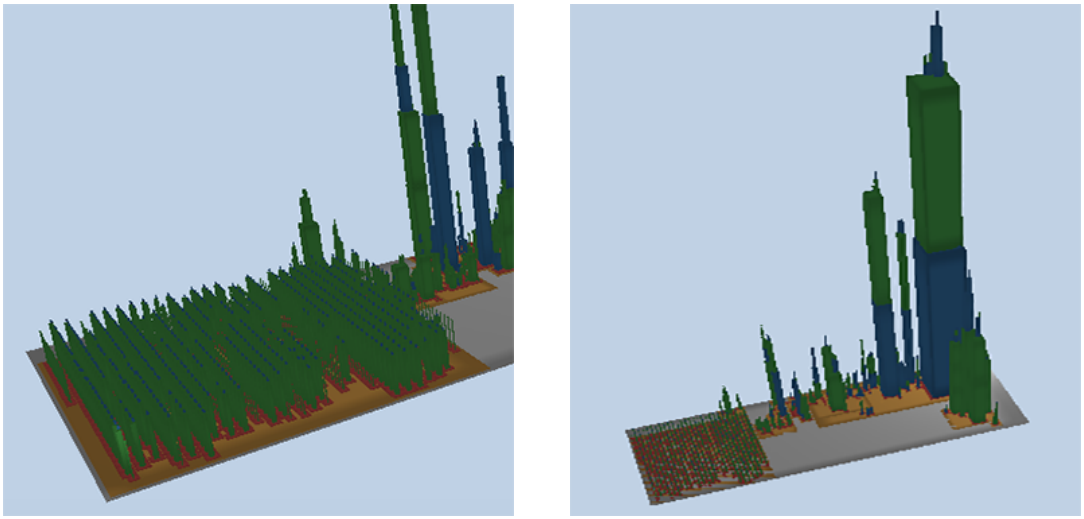


Figura 2: Diagrama de Componentes da Ferramenta JSCity

A ferramenta desconsidera na análise todos os arquivos de teste, exemplos de uso, arquivos minificados, arquivos de *copyright*, documentações e qualquer arquivo que não corresponda ao *core* do sistema em análise. Por ser uma primeira versão, o sistema possui a limitação de realizar análise de código somente em Javascript, não permitindo essa interpretação em outras linguagens de programação. Além disso, a definição das variáveis das metáforas não ocorre em tempo real, o que faz com que seja necessária uma análise prévia do que será ou não relevante para a geração das visualizações.

4. Exemplos de Uso

Seleção dos Repositórios. Para avaliar a aplicação da metáfora e solução propostas para JavaScript, foram selecionados sistemas populares escritos em JavaScript hospedados no GitHub. O critério para seleção de sistemas foi possuir uma quantidade mínima de 250 estrelas. A pesquisa ocorreu em maio de 2015 e dentre os resultados optou-se por 40 sistemas conhecidos, amplamente utilizados na web e que abrangem diferentes aplicações, tais



(a) Funções de Internacionalização (b) Funções do Núcleo

Figura 3: Cidade do sistema AngularJS

como *frameworks*, editores de código, plugins de navegadores, jogos, dentre outros. Para ilustrar o uso da metáfora de forma mais detalhada, foram selecionados três repositórios: *AngularJS, jQuery e Bower*. Os dois primeiros são *frameworks* para desenvolvimento web e o terceiro é um gerenciador de pacotes. A lista completa de repositórios analisados e suas cidades em 3D pode ser encontrada em:
`https://github.com/ASERG-UFMG/JSCity/wiki/JSCITY`.

4.1. Exemplo 1: AngularJS

A Figura 3 apresenta a cidade do sistema JavaScript mais popular no GitHub, o AngularJS. Com 39,032 estrelas, esse conhecido framework para desenvolvimento de aplicações web possui 233,785 linhas de código dividas em 20 diretórios, 863 arquivos, 10,362 funções anônimas e 6,050 funções nomeadas. Através da visualização pode-se observar duas áreas distintas: um distrito com prédios pequenos à esquerda na Figura 3a (sugerindo que contém arquivos estruturalmente similares) em contraste com arranha-céus do lado direito na Figura 3b. De fato, no diretório mostrado à esquerda estão os arquivos para internacionalização, enquanto que nos diretórios à direita estão os arquivos que fazem parte do núcleo desse sistema. Assim, a visualização proposta torna mais simples o entendimento da modularização de um sistema altamente complexo.

4.2. Exemplo 2: jQuery

A Figura 4 apresenta a cidade de um dos *frameworks* mais populares no desenvolvimento web, o jQuery. O código possui na sua maior parte funções anônimas (prédios verdes), e os grandes prédios representam funções do núcleo do *framework*, como as que tratam eventos. Esse comportamento pode ser justificado pelo uso comum de *callbacks* que são funções "passadas como argumento de outra função" e/ou chamadas quando um evento é acionado. Pode-se observar também (através das interações da visualização) que os diretórios estão separados por módulos, como por exemplo, os módulos *core*, *event*, *data*, dentre outros.

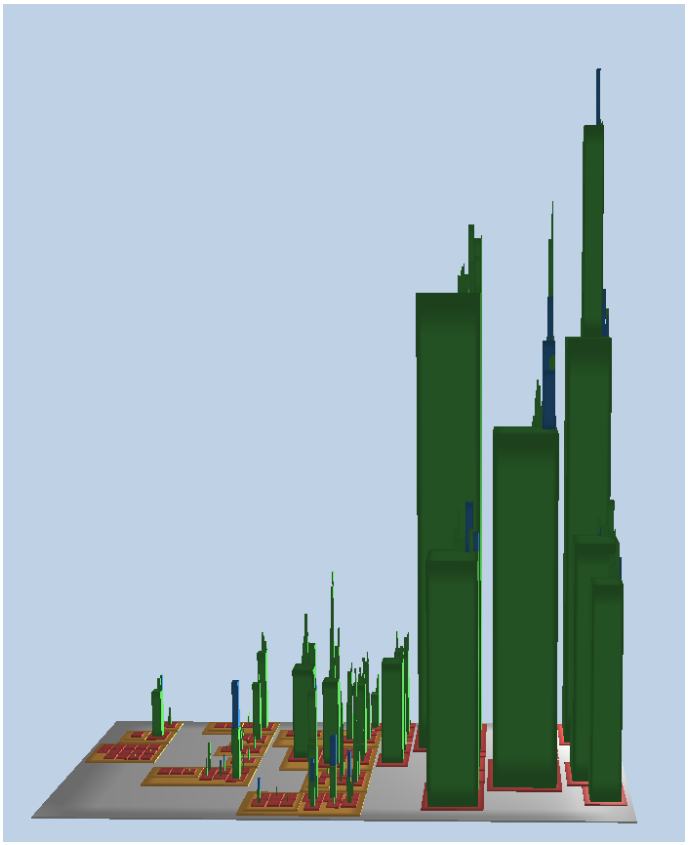


Figura 4: Cidade do sistema jQuery

4.3. Exemplo 3: Bower

A Figura 5 apresenta a cidade de um sistema JavaScript para gerenciamento de pacotes para desenvolvimento web, o Bower. Pode-se observar que não existe um diretório para internacionalização, já que esse sistema não oferece mensagens para outras línguas. Nota-se também que grande parte das funções são anônimas (prédios verdes), mas funções nomeadas também são relativamente comuns (prédios azuis). Além disso, observa-se o frequente uso de funções aninhadas (prédios sobre prédios).

Em suma, a partir da análise detalhada dos repositórios, foi possível encontrar alguns padrões de desenvolvimento, conforme sumarizado a seguir:

1. **Prédios altos e largos**: representam funções do núcleo do sistema;
2. **Distritos grandes com muitos prédios pequenos**: arquivos estruturalmente similares, por exemplo, para implementar internacionalização;
3. **Prédios verdes**: funções anônimas é um recurso da linguagem JavaScript amplamente utilizado;
4. **Prédios sobre outros prédios**: o uso de funções aninhadas é comum nos sistemas analisados, principalmente no núcleo dos sistemas.

5. Trabalhos Relacionados

Nesse trabalho, apresentou-se JSCity, inspirada no CodeCity [Wettel and Lanza 2008, Wettel et al. 2011]. CodeCity tem por objetivo a análise de software, em que sistemas

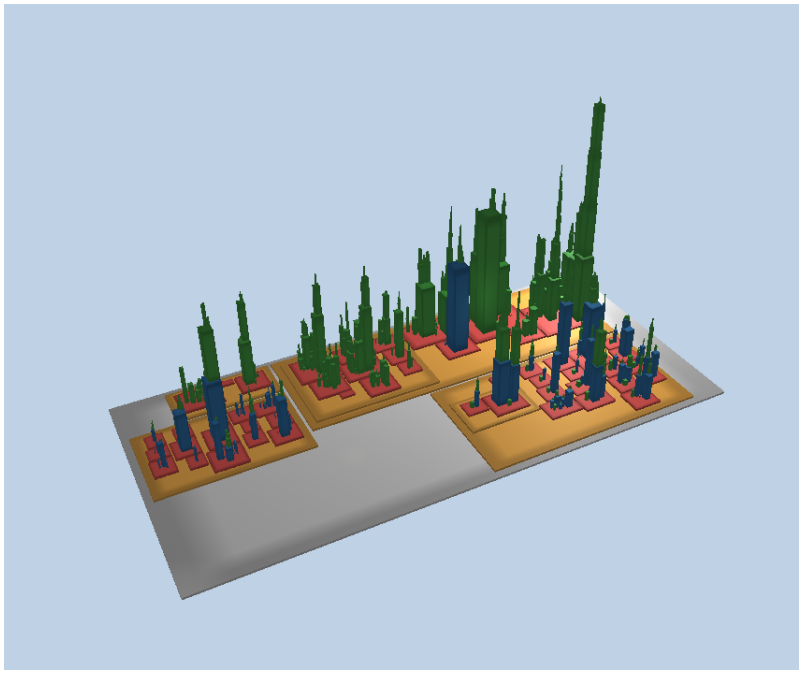


Figura 5: Cidade do sistema Bower

são visualizados como cidades 3D navegáveis e interativas. Classes são representadas como edifícios da cidade, enquanto módulos são retratados como distritos. O número de métodos representa a altura dos prédios, a quantidade de atributos representa a largura e o número de linhas de código é representado por cores - de cinza escuro (menor quantidade) a azul intenso (maior quantidade). CodeCity está disponível para a plataforma de desenvolvimento Eclipse[3] e para a ferramenta Moose[4]. No entanto, nesses casos, estão restritos a análise de sistemas orientados a objetos, nas linguagens, Java e Smalltalk. Por fim, visualizações— não necessariamente na forma de cidade—já foram propostas para outras dimensões de um sistema, como para análise de *bugs* [Hora et al. 2012].

6. Conclusões

JSCity estende CodeCity para JavaScript e oferece para a comunidade uma forma alternativa de analisar sistemas de software desenvolvidos nessa linguagem. A metáfora da cidade foi adaptada para representar sistemas JavaScript, por exemplo, através da visualização de funções anônimas e funções aninhadas. JSCity é facilmente acessível para os desenvolvedores pois roda diretamente em uma página web. As visualizações podem ser utilizadas, por exemplo, em revisões de código e para recomendação de refatorações [Sales et al. 2013, Silva et al. 2014]. Como trabalho futuro, sugere-se que seja verificada a eficácia do uso das metáforas no aumento da produtividade dos desenvolvedores. Além disso, sugere-se que sejam implementadas melhorias na ferramenta que permitam a alteração das métricas das metáforas em tempo real com o objetivo de possibilitar a geração de outros tipos de visualização.

[3]https://marketplace.eclipse.org/content/codecity
[4]http://www.inf.usi.ch/phd/wettel/codecity.html

Agradecimentos

Essa pesquisa foi apoiada pelo CNPq e FAPEMIG.

Referências

Aparecido, G., Nassau, M., Mossri, H., Marques-Neto, H., and Valente, M. T. (2011). On the benefits of planning and grouping software maintenance requests. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 55–64.

ECMA International (2011). *Standard ECMA-262 - ECMAScript Language Specification*.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of JavaScript. In *European Conference on Object-oriented Programming*.

Hidayat, A. (2012). Esprima: Ecmascript parsing infrastructure for multipurpose analysis. http://esprima.org.

Hora, A., Couto, C., Anquetil, N., Ducasse, S., Bhatti, M., Valente, M. T., and Martins, J. (2012). BugMaps: A tool for the visual exploration and analysis of bugs. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Tool Demonstration Track*, pages 523–526.

Kienle, H. M. (2010). It's about time to take JavaScript (more) seriously. *IEEE Software*, 27(3).

Nederlof, A., Mesbah, A., and Deursen, A. v. (2014). Software engineering for the web: The state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241.

Silva, D., Terra, R., and Valente, M. T. (2014). Recommending automated extract method refactorings. In *22nd IEEE International Conference on Program Comprehension (ICPC)*, pages 146–156.

Silva, L., Ramos, M., Valente, M. T., Bergel, A., and Anquetil, N. (2015). Does JavaScript software embrace classes? In *International Conference on Software Analysis, Evolution, and Reengineering*.

Stasko, J. T., Brown, M. H., and Price, B. A., editors (1997). *Software Visualization*. MIT Press.

Wettel, R. and Lanza, M. (2008). CodeCity: 3D Visualization of Large-scale Software. In *International Conference on Software Engineering*.

Wettel, R., Lanza, M., and Robbes, R. (2011). Software systems as cities: A controlled experiment. In *International Conference on Software Engineering*.

ECODroid: Uma Ferramenta para Análise e Visualização de Consumo de Energia em Aplicativos Android

Francisco Helano S. de Magalhães, Lincoln S. Rocha, Danielo G. Gomes

[1]Grupo de Redes de Computadores, Engenharia de Software e Sistemas (GREat)
Universidade Federal do Ceará (UFC)
Av. Mister Hull, s/n – Campus do Pici – Bloco 942-A
CEP: 60455-760 – Fortaleza-CE – Brasil

{helanomagalhaes,lincoln,dgomes}@great.ufc.br

Resumo. *Nos últimos anos, os dispositivos móveis evoluíram de plataformas fechadas, contendo apenas aplicações pré-instaladas, para plataformas abertas capazes de executar aplicativos desenvolvidos por terceiros. Entretanto, essa evolução trouxe consigo diversos problemas, tais como o consumo anormal de energia. Esse problema muitas vezes é causado por aplicativos mal projetados para lidar com questões de consumo de energia. Portanto, para melhorar a qualidade dos aplicativos com relação a eficiência energética, os desenvolvedores necessitam de ferramentas adequadas. Nesse contexto, este artigo apresenta o ECODroid, uma ferramenta para análise e visualização do consumo de energia em aplicativos Android que ajuda na identificação de trechos de código que possuem problemas relacionados ao consumo anormal de energia. Uma avaliação inicial do ECODroid foi realizada com o intuito de analisar a sua viabilidade.*

1. Introdução

Os dispositivos móveis (e.g., *smartphones* e *tablets*) tornaram-se parte integrante da nossa vida cotidiana. A razão para sua crescente popularidade é a possibilidade de ter um grande número de funcionalidades em um único sistema embarcado. Por meio de aplicativos móveis, esses dispositivos podem acessar a Internet, capturar fotos, capturar e reproduzir áudio e vídeo, etc. Porém, muitos desses aplicativos não levam em conta o consumo de energia quando são projetados [Pathak et al. 2012].

Estudos recentes relatam que vários aplicativos *Android* não são energeticamente eficientes devido a dois motivos principais [Liu et al. 2014]. O primeiro está relacionado ao fato do *Android* expor para os desenvolvedores APIs que possibilitam realizar operações de *hardware* (e.g., API para controlar o brilho da tela e acionar interfaces de comunicação sem fio). Embora essas APIs tragam flexibilidade para os desenvolvedores, estes devem utilizá-las com cautela, uma vez que o mau uso do *hardware* pode facilmente acarretar um alto consumo de energia. O segundo motivo está relacionado ao fato dos aplicativos *Android* serem desenvolvidos por equipes pequenas nas quais não existe um esforço dedicado às atividades de garantia da qualidade. Esses desenvolvedores raramente realizam esforços no sentido de melhorar a eficiência energética dos aplicativos.

Segundo [Liu et al. 2014], a localização de problemas de consumo anormal de energia em aplicativos *Android* é uma tarefa difícil, necessitando auxílio ferramental adequado. Nesse contexto, esse trabalho apresenta o ECODroid (*Energy Consumption Optimizer for Android*), uma ferramenta para análise e visualização do consumo de energia

em aplicativos Android. O ECODroid foi implementado como um *plugin* da IDE *Android Studio*[1] e, a partir da utilização de um modelo analítico de consumo energético, identifica as áreas do código-fonte do aplicativo que apresentam níveis anormais de consumo de energia. Uma avaliação inicial do ECODroid foi realizada, apresentando indícios de viabilidade como ferramenta de auxílio à localização de problemas de consumo de energia.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta conceitos relacionados à medição do consumo de energia. A Seção 3 apresenta o ECODroid e a Seção 4 descreve uma avaliação inicial da ferramenta. Já a Seção 5 é dedicada aos trabalhos relacionados. Por fim, a Seção 6 apresenta as considerações finais do artigo.

2. Medição do Consumo de Energia

A medição do consumo de energia de um dispositivo móvel pode ser feita de duas maneiras: *online* e *offline*. A medição *online* é realizada por meio de um dispositivo de medição externo, utilizando dispositivos de referência para os testes. Neste tipo de medição, é recomendado fazer uso de uma fonte de corrente contínua ao invés da bateria convencional. Essa abordagem ajuda a minimizar a interferência das propriedades da bateria nos valores medidos. Já a medição *offline*, fornece estimativas auferidas por meio de um software que é executado no próprio dispositivo móvel ou por meio da utilização de valores extraídos do diagnóstico de *hardware* feito pelo fabricante do dispositivo. Portanto, as medições *online* geralmente são feitas utilizando os resultados das medições *offline* como referência.

Em cenários reais, os componentes de *hardware* do dispositivo móvel não podem operar totalmente isolados, pois o resultado de cada medição é a soma do consumo de todos os componentes que estão ativos. A abordagem recomendada pelo fornecedor da plataforma *Android*[2] é subtrair da energia total consumida o valor do consumo de *standby* dos demais componentes que não estão sendo avaliados naquele momento. No entanto, há alguns componentes do dispositivo que estão sempre operantes e não podem ser simplesmente desligados, como por exemplo a CPU (*Central Processing Unit*). Obter o consumo de energia desses componentes pode ser possível por meio do cálculo algébrico de um sistema de equações lineares, consistindo na soma dos consumos de energia de um número específico de componentes e a energia total consumida pelo dispositivo em um determinado momento em um cenário de uso específico. Entretanto, esses cálculos não são necessariamente precisos e os dados obtidos devem ser validados, pois não se pode garantir que a CPU não esteja sendo usada por outros aplicativos ou por serviços que executam em *background* [Tarkoma et al. 2014].

O modelo de consumo de energia adotado neste trabalho é mesmo utilizado em [Couto et al. 2014]. Esse modelo assume que diferentes componentes de *hardware* causam diferentes impactos no consumo de energia em um dispositivo móvel e, por esse motivo, leva em consideração características particulares de cada um desses componentes. O modelo de energia utilizado no neste trabalho foi inspirado no *PowerTutor*[3] e considera seis componentes de hardware diferentes: CPU, LCD, GPS, *Wi-Fi*, 3G e Áudio.

O consumo de energia da CPU é fortemente influenciado pela sua utilização e

[1] http://developer.android.com/tools/studio/

[2] https://developer.android.com/training/monitoring-device-state/

[3] http://ziyang.eecs.umich.edu/projects/powertutor/

frequência. O processador pode trabalhar em diferentes frequências de acordo com sua necessidade. Dependendo da atividade processada, a porcentagem de utilização pode variar entre 0 e 100. O consumo de energia deste componente em um momento específico é calculado multiplicando o coeficiente associado com a frequência em uso e a porcentagem de sua utilização. Já o modelo de energia do *display* LCD considera apenas o brilho como variável. O valor do consumo é calculado baseado na porcentagem do brilho em relação aos valores de *screen.on* e *screen.full* obtidos por meio do *profile*. O consumo do GPS depende exclusivamente do seu estado atual: *gps.active*, *gps.sleep* ou *gps.off*. A interface *Wi-Fi* possui três estados de consumo no *profile* disponibilizado pelo fabricante: (i) quando o *Wi-Fi* está ligado, mas não está recebendo ou transmitindo dados (*wifi.on*); (ii) quando está transmitindo dados (*wifi.active*); e (iii) quando está procurando por pontos de acesso (*wifi.scan*). Assim como na interface *Wi-Fi*, o 3G possui três estados de consumo: os estados *radio.active* e *radio.scanning*, que são equivalentes aos estados *wifi.active* e *wifi.scan* da interface *Wi-Fi*. O terceiro estado, *radio.on*, possui dois valores de consumo, um quando está ligado, mas sem sinal e quando está ligado, porém com sinal operante. Por fim, o componente de áudio é modelado apenas por meio do estado ligado e desligado, uma vez que o volume do áudio não interfere de maneira expressiva no consumo da energia [Carroll and Heiser 2010].

3. ECODroid - *Energy Consumption Optimizer for Android*

3.1. Visão Geral

O ECODroid foi desenvolvido como um *plugin* do *Android Studio* com o objetivo de identificar regiões do código-fonte de um aplicativo Android que possuem consumo anormal de energia. A execução do ECODroid consiste em três etapas: (i) instrumentação; (ii) execução e teste; e (iii) visualização de resultados. Na etapa de instrumentação, o código-fonte do aplicativo a ser analisado passa por um processo de instrumentação. No fim desta etapa, uma cópia do projeto contendo o código instrumentado é criada e armazenada em um diretório chamado "`_INSTRUMENTED_`". Já na etapa de execução e teste, a cópia do projeto criada na etapa anterior é compilada e executada de forma automática a fim de realizar a análise de consumo de energia. Por fim, na etapa de visualização dos resultados, os dados sobre consumo de energia gerados na etapa anterior são analisados para determinar quais regiões do código do aplicativo são responsáveis pelo o consumo anormal de energia. O resultado dessa análise é apresentado ao desenvolvedor no formato de um gráfico *Sunburst*[4]. Cada uma dessas etapas são detalhadas nas próximas seções.

3.2. Instrumentação do Aplicativo

O processo de instrumentação no ECODroid é feito de forma automática com o auxílio da biblioteca *JavaParser*. Essa instrumentação consiste na inserção de chamadas à API do ECODroid responsável pela aquisição do consumo de energia. Essa API é uma versão adaptada do *framework* desenvolvido em [Couto et al. 2014]. A API possui a classe `Estimator` que provê a implementação dos métodos cujas chamadas são inseridas no código-fonte do aplicativo via instrumentação. Os métodos utilizados na instrumentação do aplicativo são: `traceMethod()`, responsável por rastrear o consumo de energia dos

[4] *Sunburst* é um desenho ou figura constituída por raios ou "vigas" que irradiam para fora a partir de um disco central em forma de raios de sol.

métodos da aplicação; `saveResults()`, responsável por salvar os resultados do perfil de energia em um arquivo; `start()`, responsável por iniciar a *thread* de monitoramento de energia; e `stop()` responsável por parar a *thread* de monitoramento de energia.

No processo de instrumentação, todos os métodos das classes do aplicativo, assim como as classes destinadas aos testes, terão novas linhas de código adicionadas. Essas linhas de código são inseridas no começo (logo após a declaração do método) e no final (antes da instrução `return` ou na última instrução para métodos que retornam `void`) de cada método. A Figura 1 apresenta um exemplo de código instrumentado. As linhas de código 4 e 6 fazem chamadas à API do ECODroid para inicializar e finalizar, respectivamente, o rastreamento e medição do consumo de energia do método `initialize()` da classe `Controller`.

```
1. public class Controller {
2.  ...
3.  public void initialize(){
4.     Estimator.traceMethod("initialize", "Controller", Estimator.BEGIN);
5.     ...
6.     Estimator.traceMethod("initialize", "Controller", Estimator.END);
7.  }
8.  ...
9. }
```

Figura 1. Exemplo de método instrumentado.

O ECODroid utiliza o *framework* de testes do *Android* para executar automaticamente a aplicação a ser analisada, assim como aplicar os diferentes casos de teste. A API de testes do *Android* é baseada no *JUnit*, a qual é parte integrante do ambiente de desenvolvimento, fornecendo uma arquitetura e ferramentas que ajudam o desenvolvedor a testar seu aplicativo. O processo de instrumentação também é aplicado na classe que possui os casos a serem testados. Os métodos `setUp()` e `tearDown()` são adicionados para serem executados no começo e no final de cada caso de teste, respectivamente, como exemplificado no código da Figura 2.

```
1. public class MyTestCase extends TestCase {
2.  ...
3.  public void setUp(){
4.     Estimator.start(uid);
5.     ...
6.  }
7.  ...
8.  public void tearDown(){
9.     Estimator.stop();
10.    ...
11. }
12. ...
13.}
```

Figura 2. Exemplo de código de teste instrumentado.

Com esta abordagem, o método `Estimator.start(uid)` (linha 4, Figura 2) é chamado toda vez que o caso de teste é iniciado. Nesse momento, o ECODroid inicia a *thread* que coleta as informações do sistema operacional para o cálculo do consumo de energia e em seguida aplica o modelo analítico para estimar a demanda energética. O argumento `uid` do método `start()` é o *Android* `UID` (*Unique Identification*) do aplicativo em teste, o qual é necessário para coletar as informações corretamente. Dentro

do método `tearDown()` é feita uma chamada ao método `Estimator.stop()` (linha 9, Figura 2) que finaliza a execução da *thread* de monitoramento, salvando os resultados obtidos em um arquivo.

3.3. Execução e Teste

Nesta etapa, todas as classes do aplicativo e os seus casos de teste já se encontram instrumentados. Desse modo, um arquivo contendo um conjunto de comandos executados em lote é criado para automatizar o processo. O primeiro comando a ser executado é o `android update project`, o qual é utilizado para atualizar configurações do projeto, como versão do SDK, ambiente de desenvolvimento, modificar o *target* e o nome do projeto. Após sua execução será gerado todos os arquivos e diretórios que estão faltando ou estão desatualizados. O comando tem a seguinte sintaxe:

```
> android update project --name <project_name> --target <target_id> --path <project_path
    >
```

Em seguida são executados os comando `ant clean` e `ant debug`, utilizados para limpar o projeto e criar módulos de depuração, respectivamente.

```
> ant <path_to_your_project> build.xml clean
> ant <path_to_your_project> build.xml debug
```

Para instalar a aplicação e os casos de teste no dispositivo é utilizado o comando `adb install`. O ADB faz parte do pacote do SDK do *Android* e possibilita a comunicação entre o computador e o dispositivo móvel, tornando possível a instalação de aplicações, a troca de dados e também a execução comandos de `shell`.

```
> adb install <path_to_application_apk>
> adb install <path_to_test_apk>
```

Os testes são executados por meio do comando `adb shell am instrument`. Cada caso de teste é executado duas vezes, utilizando opções e `flags` diferentes. No comando abaixo, `<test_package>` representa o nome do pacote de teste atribuído no arquivo do `AndroidManifest.xml`, enquanto `<runner_class>` representa o nome da classe do executor de testes que está sendo utilizado, no caso do ECODroid, o `InstrumentationTestRunner`.

```
> adb shell am instrument [flags] <test_options> <test_package>/<runner_class>
```

Por fim, o comando `adb pull` é executado para que os dados de consumo de energia obtidos sejam copiados do dispositivo móvel (`<remote_folder_path>`) para o computador que está sendo utilizado (`<local_folder_path>`) pelo desenvolvedor.

```
> adb pull <remote_folder_path> <local_folder_path>
```

3.4. Visualização dos Resultados

A Figura 3 apresenta a interface gráfica do ECODroid. Nela é possível identificar duas *Tool Windows* distintas: (i) a **janela principal** (assinalada com o número 1), que apresenta um gráfico *Sunburst* que permite visualizar a distribuição do consumo de energia sobre o código do aplicativo analisado; e (ii) a **janela secundária** (assinalada com o número 2), que traz detalhes sobre o consumo de energia do aplicativo analisado.

O gráfico *Sunburst* apresentado na janela principal do *plugin* retrata dados hierárquicos a partir de discos radiais. No ECODroid, o disco central representa o projeto analisado e o disco radial seguinte refere-se aos pacotes do projeto. As classes são mapeadas no disco radial seguinte e, por fim, os métodos são mapeados no disco radial mais

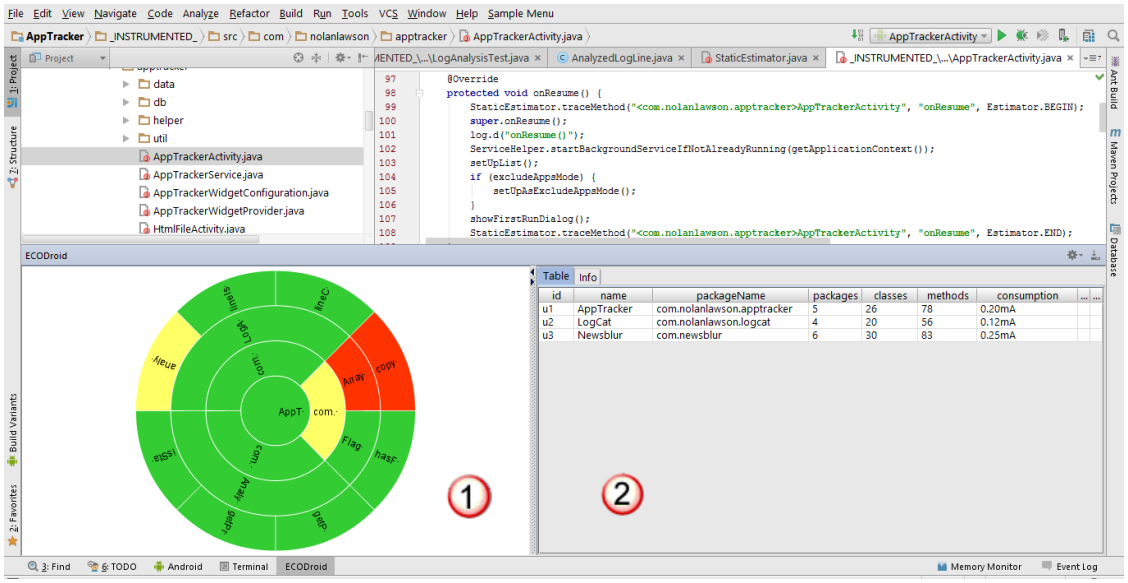


Figura 3. Interface gráfica do ECODroid

externo. Além disso, de acordo com a influência no consumo de energia, o método pode ser classificado como "verde", "amarelo" ou "vermelho". Os 'métodos verdes" são métodos que não influenciam no consumo anormal de energia, dificilmente são chamados quando o aplicativo consome energia acima da média. Já os "métodos vermelhos" influenciam significativamente para o consumo anormal de energia. Com base no modelo analítico adotado, uma aplicação para ser classificada como de baixo consumo de energia deve possuir no máximo 30% dos seus métodos classificados como " métodos vermelhos". Por fim, os "métodos amarelos" possuem influência intermediária no consumo de energia. Essa representação de métodos é extensível para classes, pacotes e projetos. Assim, se uma classe possui mais de 50% dos seus métodos classificados como "vermelhos", então ela também é considerada uma "classe vermelha". Por outro lado, se possui mais de 50% de seus métodos classificados como "verdes", trata-se de uma "classe verde".

A janela secundária está subdividida em duas abas, `Table` e `Info`. A aba `Table` é responsável por mostrar o histórico de informações de projetos que foram analisados ao longo do tempo e possui os seguintes campos: nome do projeto, nome do pacote principal do projeto, quantidade de pacotes, classes e métodos do projeto, além de fornecer o consumo de energia obtido com a análise da ferramenta. Já a aba `Info` permite ao desenvolvedor visualizar as informações sobre o componente selecionado no gráfico *Sunburst*. Quando o desenvolvedor "clica" em alguma parte do gráfico, as seguintes informações sobre o elemento selecionado são apresentadas: o nome do elemento, o seu tipo (projeto, pacote, classe ou método), os nomes dos elementos que estão na hierarquia acima (se houver), o impacto no consumo da aplicação e o consumo medido em joules.

4. Avaliação Inicial

Até o presente momento nenhuma avaliação rigorosa (e.g., estudo de caso ou experimento controlado) foi conduzida para avaliar a real eficácia do ECODroid. Entretanto, um conjunto de aplicativos móveis, desenvolvidos em parcerias do nosso

grupo de pesquisa[5] com empresas de grande porte do setor de mobilidade, vêm sendo alvo de avaliações de consumo de energia por meio do ECODroid. Essas avaliações têm sido úteis para identificar áreas do código-fonte desses aplicativos que possuem problemas relacionados ao consumo anormal de energia. De posse da localização dos trechos de código mais críticos, têm sido possível concentrar esforços para reduzir esses problemas. Durante essas avaliações, foi possível identificar que existe um método específico da API padrão do *Android* que, quando invocado dentro de um aplicativo, faz aumentar consideravelmente o consumo médio de energia. Trata-se do método `dispatchMessage(android.os.Message)` da classe `android.os.Handler`, dedicado a tratar mensagens de sistema.

Com base nessa descoberta, foi selecionado um aplicativo exemplo para evidenciar a identificação dessa anomalia de consumo de energia com o uso do ECODroid. O aplicativo escolhido como alvo dessa avaliação foi o `App Tracker`, que possui código fonte aberto e disponível para *download* no `GitHub`[6]. Esse aplicativo registra as estatísticas de uso de outros aplicativos por meio de uma tarefa que executa em *background* e exibe essas informações em uma interface gráfica (*widget* ou na *activity* principal). Dessa forma, foram feitas duas avaliações do `App Tracker` utilizando o ECODroid. Na primeira, o aplicativo foi avaliado sem que nenhuma alteração tenha sido feita no seu código fonte. Já na segunda avaliação, uma chamada ao método `dispatchMessage()` da API padrão do *Android* foi inserida no código do `App Tracker`. As Figuras 4 e 5 apresentam, respectivamente, o gráfico *Sunburst* para a primeira e para a segunda avaliação.

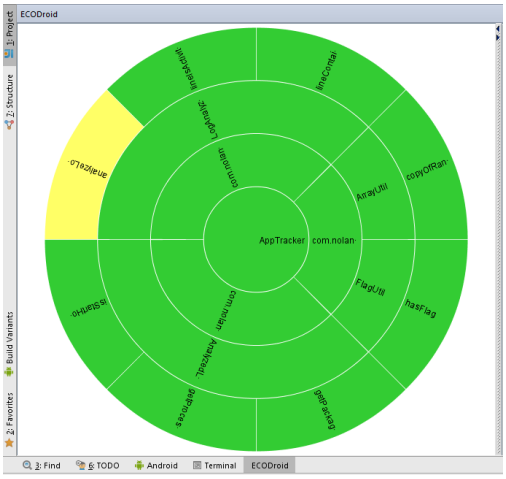


Figura 4. Consumo de energia do `App Tracker` original.

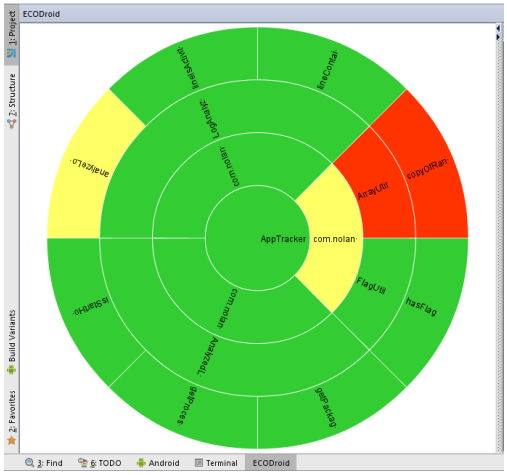


Figura 5. Consumo de energia do `App Tracker` após alteração.

Observe que para a primeira avaliação (Figura 4) apenas o método `analyzeLogLine` obteve consumo de energia um pouco superior a média e foi classificado como sendo um "método amarelo". Já na segunda avaliação (Figura 5), o método `copyOfRange`, que inicialmente é classificado como "método verde" na primeira avaliação (Figura 4), foi modificado para fazer chamadas ao método `dispatchMessage` da API padrão do *Android*. Com essa alteração, o método passou a ser classificado como "método vermelho", evidenciando assim o aumento no consumo de energia.

[5]http://www.great.ufc.br/

[6]https://github.com/nolanlawson/AppTracker

5. Trabalhos Relacionados

O trabalho descrito em [Couto et al. 2014] é fortemente relacionado ao ECODroid. Tanto o modelo analítico de consumo de energia, quanto a forma de apresentar os resultados no formato de gráfico de *Sunburst*, serviram de inspiração para este trabalho. Entretanto, diferente de [Couto et al. 2014], o ECODroid utiliza valores de referência, providos pelos fabricantes, sobre o consumo energético dos componentes de hardware do dispositivo, tornando as suas estimativas mais precisas. Além disso, o ECODroid provê ao desenvolvedor um suporte ferramental automatizado e integrado ao ambiente de desenvolvimento.

Já o trabalho descrito em [Liu et al. 2014] apesenta um estudo sobre a identificação das causas de *bugs* relacionados ao consumo anormal de energia em aplicativos *Android*. Naquele trabalho, os autores utilizam uma técnica de instrumentação de código para gerar *traces* de execução que serão analisados por meio de um verificador de modelos. Essa análise ajuda a identificar padrões de *bugs* previamente catalogados. Enquanto o trabalho de [Liu et al. 2014] está preocupado em encontrar a causa dos problemas de consumo de energia, o ECODroid está preocupado em identificar as partes do código-fonte que apresentam consumo anormal de energia. Portanto, o trabalho de [Liu et al. 2014] pode ser visto como uma abordagem complementar do presente trabalho.

6. Considerações Finais

Este trabalho apresentou o ECODroid, uma ferramenta para análise e visualização do consumo de energia em aplicativos Android. Uma avaliação inicial do ECODroid foi realizada, apresentando indícios de sua viabilidade como ferramenta de auxílio na localização de trechos de código-fonte que apresentam consumo anormal de energia. Embora o ECODroid ajude a localizar o trechos de código-fonte problemáticos, ela não oferece suporte à resolução dos problemas de consumo de energia identificados, sendo esse um dos trabalhos futuros a ser realizado. Além disso, uma avaliação mais rigorosa do ECODroid deve ser conduzida a fim de evidenciar a sua efetividade.

Referências

Carroll, A. and Heiser, G. (2010). An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, Berkeley, CA, USA. USENIX Association.

Couto, M., Carção, T., Cunha, J., Fernandes, J., and Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. In Quintão Pereira, F. M., editor, *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 77–91. Springer International Publishing.

Liu, Y., Xu, C., Cheung, S., and Lu, J. (2014). Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 40(9):911–940.

Pathak, A., Hu, Y. C., and Zhang, M. (2012). Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA. ACM.

Tarkoma, S., Siekkinen, M., Lagerspetz, E., and Xiao, Y. (2014). *Smartphone Energy Consumption: Modeling and Optimization*. Cambridge University Press.

VisMinerService – A REST Web Service for Source Mining

Luis Paulo da Silva Carvalho[1,2] , Renato Novais[1,3], Manoel Gomes de Mendonça Neto[2]

[1]Federal Institute of Bahia (IFBA)

[2]Federal University of Bahia (UFBA)

[3]Centro de Projeto Fraunhofer para Engenharia de Software e Sistemas at UFBA
{luiscarvalho, renato}@ifba.edu.br, manoel.mendonca@ufba.br

Abstract. *Enabling applications to perform source-mining and software metrics visualization can be a time-consuming task. With this in mind, we are positive that Web Services are fitting candidates to automate the developing of such applications. Therefore, this paper presents VisMinerService, a REST Web Service implementation built upon a static software component, VisMiner. It is described how the service can be used in order to mine metrics and other source-code related information. It is also discussed its integration with three different types of client applications (mobile, desktop/swing and web application) in order to provide a proof of concept of how VisMinerService can be used to create platform-independent software to visualize the mined information.*

1. Introduction

Manual collecting of metrics is an unreliable activity, since "too many errors or missing data badly affect the analysis process" [Sillitti et al. 2003]. Thus, providing ways to automate the execution of source-mining is an important activity. Plus to that, once the source-code is mined, it is necessary to portray it in a way that improves the understanding of the information, so to later aid software engineers to carry development tasks (e.g. maintenance, refactoring, and reverse engineering).

In order to better assist source-mining and visualization tasks, we consider that is important to fulfill the following objectives: (a) smoothing the effort to retrieve source code and metrics information in a way that one could easily build applications to make use of it (e.g. to either analyze or visualize the data); (b) automating the outsource of the gathered information in a well defined way, so that it becomes easier to parse and visualize the mined information; (c) grating the outsourcing in a distributed way (e.g. as with the use of Web Services), one could easily manage to create client software to reuse the routines to mine source-codes and metrics remotely.

Web Services have become a standard for sharing data between software applications over the internet. They enable the creation of technology-neutral, open, decentralized, reliable software solutions [Pautasso 2014]. Web Services provide distributed functionalities, which are independent of hardware platform, operating system and programming language. An example of Web Services implementation, which has increasingly gained acceptance due to its simpler style to use, is REST Web

Services [Al-Zoubi and Wainer 2009]. REST[1] is a Web Services' architectural style that constrains the interface of the HTTP protocol to a set of standard operations (GET, POST, PUT, and DELETE).

REST is a fitting candidate to implement service-oriented applications with flexibility and lower overhead [Hamad et al 2010]. In the same way, it can be used on the context of source-mining, metrics calculation, and visualization. However, either few works have explored REST to supply access to metrics using visualization or, if any, the potentials of using REST services in such field of application have not yet been experimented to its fullest.

VisMiner is a core library that comprises a set of functionalities dedicated to the extraction of information from source code files [Mendes et al. 2015]. It enables third-party applications to mine metrics from on-line Version Control Systems (VCS) as, for example, GitHub. So far, VisMiner has comprised the following set of metrics: Number of Lines of Code (LOC), Ciclomatic Complexity (CC), Number of Packages (NOP), and Number of Methods (NOM).

This work describes an approach by which the VisMiner Library is used as a component to create a REST Web Service, called VisMinerService. The goal is to externalize Vismine's functionalities across the web to enable the development of distributed service-oriented applications and toolsets to mine and visualize software data. We have also conducted a preliminary test in the development of multiple clients. The goal is to show the advantages of a service-oriented architecture for source-mining, metrics calculation, and visualization.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the VisMinerService. Section 4 exemplifies the use of VisMinerService. Conclusions and future works are discussed in Section 5.

2. Related work

The use of Web Service-based architectures to mine and visualize metrics obtained from software repositories is not new. Sillitti et al (2003) investigated the application of XML and SOAP (Simple Object Access Protocol) to expose functionalities in order to aid the extraction of metrics across HTTP-enabled networks. SOAP also makes possible the creation of Web Services. However, as previously stated, REST has later become a more adopted standard.

Sakamoto et al (2013) proposed the MetricWebAPI, which wraps mechanisms within a service-oriented framework to mine metrics. It also externalizes data to client applications via XML documents. However, it is not provided information about how such client applications can reuse the functionalities of MetricWebAPI. Protocols and document formats to base the interactions between the service and client applications are not covered.

Several other projects are intended to mine source-codes, but they either mention little (or no) remarks on how to be reused by client applications or there is no evidence that they were design for reuse at all. In this category we could mention: Metrics

1 Short form of REpresentational State Transfer

Grimoire [METRICSGRIMOIRE, 2015], GHTorrent project [GHTORRENT, 2015], SourceMiner [SOURCEMINER, 2015].

3. VisMinerService

VisMinerService is a REST Web Service built on the top of VisMiner core library. Its main purpose is to offer functionalities to remote distributed client applications, enabling the sharing, processing and visualization of software's mined data. Figure 1 depicts the targeted usage scenario for VisMinerService.

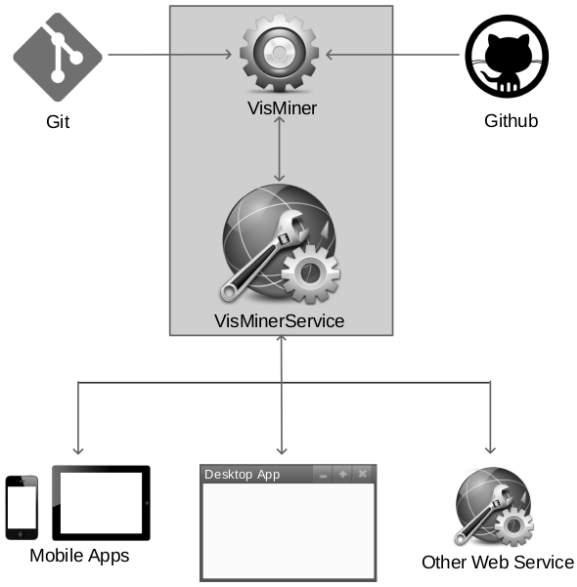


Figure 1. VisMinerService's use scenario

We have embedded VisMinerService with a set of URLs, each one triggering specific retrievals from the mined information. Table 1 lists examples of VisMinerService's externalized functionalities. The domain was omitted from the URLs, because it may vary depending on how the service's server is configured.

Table 1. VisMinerService's current URLs set

URL	Purpose
http://...projects/byid	It retrieves information about a project identified by its ID
http://...committers/byproject	It returns all committers from a given project
http://...commit/byproject	It retrieves information about all commits of a given project
http://...commits/bycommitter	It retrieves commits of a specific committer
http://...commits/numberofbycommitter	It returns the number of commits of a specific committer
http://...commits/bysha	It retrieves a commit identified by its SHA[2]
http://...file/bycommit	It fetches files affected by a specific commit
http://...metrics/byfile	It retrieves all metrics of an identified file
http://...metrics/bycommit	It returns metrics pertaining to a file modified by a commit

2 Secure Hash Algorithm is used by GitHub to identify commits

In the example shown in Figure 2, the "Complexity By Commit" URL filters data out after being parameterized by a commit's identification key. The commit was persisted in VisMiner's database at the moment that project's data (from TOMCAT project) was crawled from its GitHub repository. The retrieved JSON document represents a collection of key-value pairs. In the specific case of the Ciclomatic Complexity (CC) metric, key-value pairs are used to associate each file's methods to their respective complexities (e.g. the method "fromHexString" has summed up a CC's value of 3). Same as manually done on a web browser, computational agents (i.e. client software) can access the VisMinerService's URLs programmatically. By obtaining and parsing the JSON document, one can uses the information about the metric (and the associated file) for any specific desired purpose.

Figure 2. JSON representation of Ciclomatic Complexity (CC)

Client applications (e.g. Mobile Apps, Desktop Applications, and other Web Services) can reach VisMinerService via a set of URLs (contained in Table 1) in order to gain access to its remote functionalities. In this case, any software capable of navigating over the HTTP can be a client. Figure 2, for instance, shows a web browser (Google Chrome Web Browser) receiving and displaying a JSON [JSON, 2015] document from the *http://...metrics/complexitybycommit* URL. This URL is an entry-point to the service's functionality that retrieves information about metrics related to files affected by a particular commit.

Section 4 presents three examples of service-oriented applications. The purpose is to illustrate VisMinerService as a permissive tool that enables the development of remote client software. The use of the aforementioned URLs (Table 1) is also exemplified.

4. VisMinerService's multiple-client scenarios of use

In this section we show how different client applications can use VisMinerService. The purpose is: (i) to reinforce that VisMinerService is platform-independent; and (ii) to

show that client software can perform a multitude of processing on the data obtained from VisMinerService.

4.1. VisMinerDroid

VisMinerDroid is an ANDROID application that makes use of VisMinerService to present data visualizations to mobile users. This example comprises two screen fragments (Figure 3):

- Committers list (left side of Figure 3) → it represents a list containing all committers from a project retrieved by the mobile application after navigating to the *http://...committers/byproject* URL;
- Commits per Committer (right side of Figure 3) → it plots a pie chart to display the percentage of commits sent by committers selected from the list. The mobile application browses the *http://...commits/numberofbycommitter* URL to obtain the total number of commits authored by each committer. The resulting dataset is rendered out to a chart. ANDROIDPLOT API [AndroidPlot, 2015] was used to draw the charts.

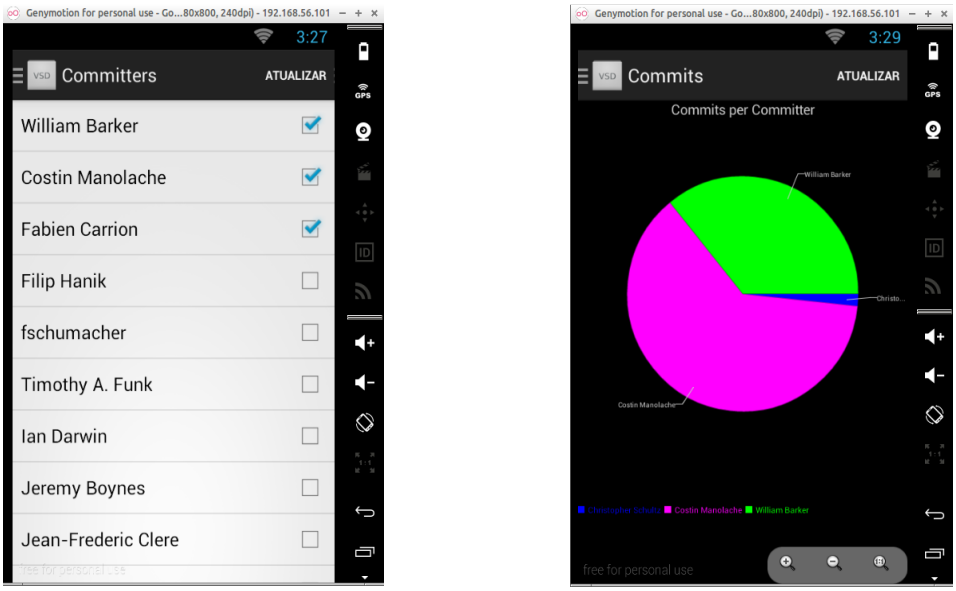


Figure 3. VisMinerDroid

VisMinerDroid is an example of how project-related information can be displayed by android applications. The advantage is to follow the global tendency of software becoming resident in mobile devices, without requiring that such applications retains all the data it might need in local databases. The information can be retrieved remotely from VisMinerService as soon as it becomes necessary.

4.2. VisMinerSwing

VisMinerSwing is a JAVAX Swing Application intended to show software-related data to desktop users. In combination with PREFUSE [PREFUSE, 2015] it shows an

interactive tree, which enables the navigation through the data received from VisMinerService. Figure 4 shows a partial view of VisMinerSwing.

VisMinerSwing navigates to the *http://...committers/byproject* URL of VisMinerService to retrieve all committers from the project. After the user chooses one committer, VisMinerService is contacted again via the *http://...commits/bycommitter* URL to specifically select the commits belonging to the committer. The commits are then distributed across the tree through a stratification of temporal subsections (commits per year, month, day, time of the day) as shown on the left side of Figure 4. The right side of Figure 4 shows the files modified by a particular commit and the corresponding accumulated values of the metrics.

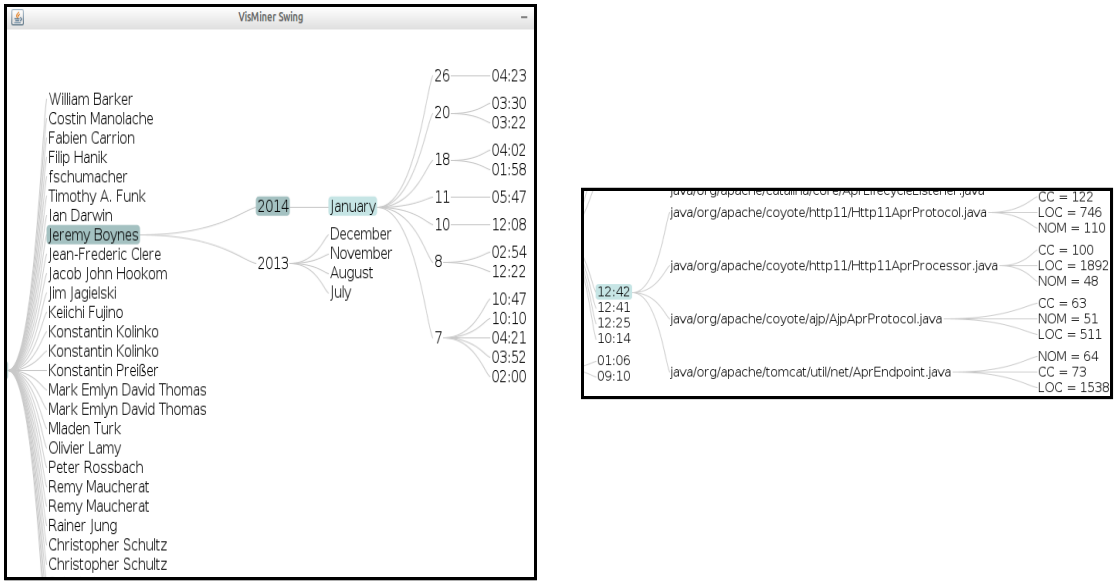


Figure 4. VisMinerSwing's commits-related information

VisMinerSwing and VisMinerDroid share the *http://...committers/byproject* URL. This is an example of how different types of application can use VisMinerService to exhibit the same information, i.e. desktop and mobile users may get access to the same information through a varied combination of visualizations. Therefore, it is possible to evidence that VisMinerService contributes to the fulfillment of sharing the use of project-related information in a platform-independent manner.

4.3. VisMinerWebViewer

The third example of VisMinerService client is a web application that enables the visualization of commits-related data via web browsers. Figure 5 presents VisMinerWebViewer. VisMinerWebViewer calls the *http://...commits/bycommitter* URL to capture commits from a chosen committer. In the example (left side of Figure 5), a timeline visualization from VIS.JS [VISJS, 2015] distributes the commits over a period of time. Users can navigate on the timeline and select a commit. After selecting a commit, another visualization based on GOOGLE CHARTS API [GCHARTS, 2015] is shown. Right side of Figure 5 contains a gauge-like visualization that exhibits metrics obtained from files affected by the commit. The set of files is extracted from

VisMinerService's response to navigations to the *http://...metrics/bycommit* URL.

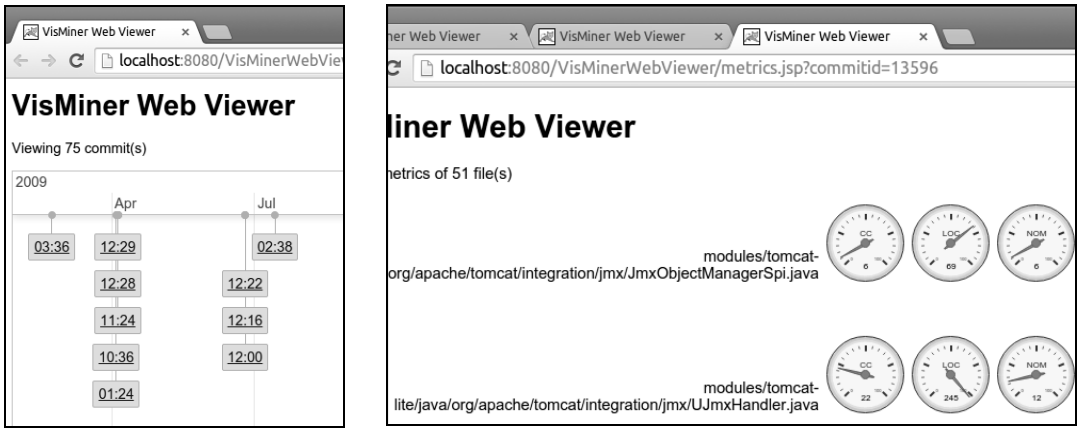


Figure 5. VisMinerWebViewer's timeline of commits

5. Conclusion and future work

In this paper, we have presented VisMinerService, a REST Web Service that intends to decentralize the extraction and visualization of source-code information. The service is capable of outputting information in a reusable and platform-independent way. We were able to prove the concept behind VisMinerService by creating different types of software, which were fully capable of showing the information gathered from the service. Such approach is highly recommended, provided that there is global tendency of information becoming pervasive [Resmini and Rosati 2011]. In this sense, source-code data (e.g. software metrics) might have to find their way through a plethora of platforms and visualization solutions while requiring either little or no adaptation at all. A service-oriented architecture is a powerful candidate to enable this.

In spite of having found strong evidences with regard to the aforesaid advantages, a more elaborated study must be conducted, because it is necessary to precisely measure the gain VisMinerService grants to the development of applications that share and visualize information mined from repositories. The experiment described in [Kaneshima et al. 2013] might be applicable in this case.

Furthering on the development of VisMinerService, we must expand the options of mined information (e.g. by adding metrics). As a consequence, more JSON documents and URLs must be created to allow new outputs. Efforts must also be made in order to automate the mining of software projects from other Version Control Systems. Ideally, VisMiner should not be limited to Git/Github's only. We must also keep the service up to date with the latest Visminer's additions: (i) inclusion of new metrics (e.g. WMC or Weighted Method Count); (ii) processing of varied targeted languages (C++ and JAVA projects) and (iii) new visualizations to externalize the new types of data that comes with such new additions.

It is also necessary to furnish end clients with visualization mechanisms; which can be achieved by increasing VisMinerService's URLs with functionalities to automate the adoption of visualization APIs. The purpose is to reduce the effort required to create graphical interfaces to show the mined data.

The integration of VisMinerService and the client applications exemplified in this work is also shown in the captured video: https://youtu.be/MOsNp45fS7c.

References

Al-Zoubi, K., & Wainer, G. (2009, June). “Using REST web-services architecture for distributed simulation”. In Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation (pp. 114-121). IEEE Computer Society.

GCHARTS (2015), https://developers.google.com/chart/. Acessed 2015, May.

GHTORRENT (2015), http://ghtorrent.org/. Acessed 2015, May.

Hamad, H., Saad, M. and Abed, R. (2009) “Performance Evaluation of RESTful Web Services for Mobile Devices”, In: International Arab Journal of e-Technology, Vol. 1, No. 3, January 2010.

JSON (2015), http://www.json.org/. Acessed 2015, May.

Kaneshima, Eliana, Maria Cristina Neves de Oliveira, and Rosana Teresinha Vaccare Braga. "Avaliação da Integração de Aplicações Empresariais usando Web Services: um Estudo Empírico". In X Workshop de Manutenção de Software Moderna, 2013, Salvador - BA. Anais do WMSWM 2013, 2013. v. 11. p. 1-8.

Mendes, T. S.; Almeida, D.; Alves, N.S.R; Spínola, R.O. ; Novais, R.L. ; Mendonça, M. “VisMinerTD – An Open Source Tool to Support the Monitoring of the Technical Debt Evolution using Software Visualization”. In: 17th International Conference on Enterprise Information Systems (ICEIS), 2015, Barcelona.

MetricsGrimoire (2015). http://metricsgrimoire.github.io/. Acessed 2015, May.

Pautasso, Cesare. "RESTful web services: principles, patterns, emerging technologies." Web Services Foundations. Springer New York, 2014. 31-51.

PREFUSE (2015), http://prefuse.org/. Acessed 2015, May.

Resmini, Andrea, and Rosati, Luca. “Pervasive Information Architecture: Design Cross-Channel User Experiences”. In IEEE Transactions on Professional Communication, Vol. 54, No. 4. Pp 408-409. IEEE, 2011.

Sakamoto, Yasutaka, Shinsuke Matsumoto, Sachio Saiki, and Masahide Nakamura. "Visualizing Software Metrics with Service-Oriented Mining Software Repository for Reviewing Personal Process." In Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on, pp. 549-554. IEEE, 2013.

Sillitti, A.; Janes, A.; Succi, G.; Vernazza, T. “Collecting, integrating and analyzing software metrics and personal software process data”. Euromicro Conference, 2003. Proceedings. 29th, vol., no., pp.336,342, 1-6 Sept. 2003.

SOURCEMINER (2015), http://www.sourceminer.org/. Acessed 2015, May.

VISJS (2015), http://visjs.org. Acessed 2015, May.

Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems

Alessandra Levcovitz[1], **Ricardo Terra**[2], **Marco Tulio Valente**[1]

[1]Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil

[2]Universidade Federal de Lavras (UFLA), Lavras, Brazil

alessandralev@gmail.com,terra@dcc.ufla.br,mtov@dcc.ufmg.br

Abstract. *The idea behind microservices architecture is to develop a single large, complex, application as a suite of small, cohesive, independent services. On the other way, monolithic systems get larger over the time, deviating from the intended architecture, and becoming tough, risky, and expensive to evolve. This paper describes a technique to identify and define microservices on a monolithic enterprise system. As the major contribution, our evaluation demonstrate that our approach could identify good candidates to become microservices on a 750 KLOC banking system, which reduced the size of the original system and took the benefits of microservices architecture, such as services being developed and deployed independently, and technology independence.*

1. Introduction

Monolithic systems inevitably get larger over the time, deviating from their intended architecture and becoming hard, risky, and expensive to evolve [6, 8]. Despite these problems, enterprise systems often adopt monolithic architectural styles. Therefore, a major challenge nowadays on enterprise software development is to evolve monolithic system on tight business schedules, target budget, but keeping quality, availability, and reliability [3].

Recently, microservices architecture emerged as an alternative to evolve monolithic applications [2]. The architecture proposes to develop a system as a set of cohesive services that evolve over time and can be independently developed and deployed. Microservices are organized as a suite of small services where each one runs in its own process and communicates through lightweight mechanisms. These services are built around business capabilities and are independently deployed [5]. Thereupon, each service can be developed in the programming language that suits better the service characteristics, can use the most appropriate data persistence mechanism, can run on an appropriate hardware and software environment, and can be developed by different teams. There are many recent success stories on using microservices on well-known companies, such as Amazon and Netflix [7].

Therefore, microservices are an interesting approach to incrementally evolve enterprise application. More specifically, new business functions can be developed as microservices instead of creating new modules on the monolithic codebase. Moreover, existing components can be extracted from the monolithic system as microservices. This process may contribute to reduce the size of monolithic application, and create smaller and easier to maintain code.

In this paper, we describe a technique to identify microservices on monolithic systems. We successfully applied this technique on a 750 KLOC real-world monolithic banking system, which manages transactions from 3.5 million banking accounts and performs nearly 2 million authorizations per day. Our evaluation shows that the proposed technique is able to identify microservices on monolithic system and that microservices can be a promising alternative to modernize legacy enterprise monolithic system.

The remainder of this paper is organized as follows. Section 2 describes the proposed technique to identify microservices on monolithic systems. Section 3 evaluates the proposed technique on a real-world monolithic banking system. Section 4 presents related work and Section 5 concludes the paper.

2. The Proposed Technique

The proposed technique considers that monolithic enterprise applications have three main parts [6]: a *client side* user interface, a *server side* application, and a *database*, as shown in Figure 1. It also considers that a large system is structured on smaller subsystems and each subsystem has a well-defined set of business responsibilities [1]. We also assume that each subsystem has a separate data store.

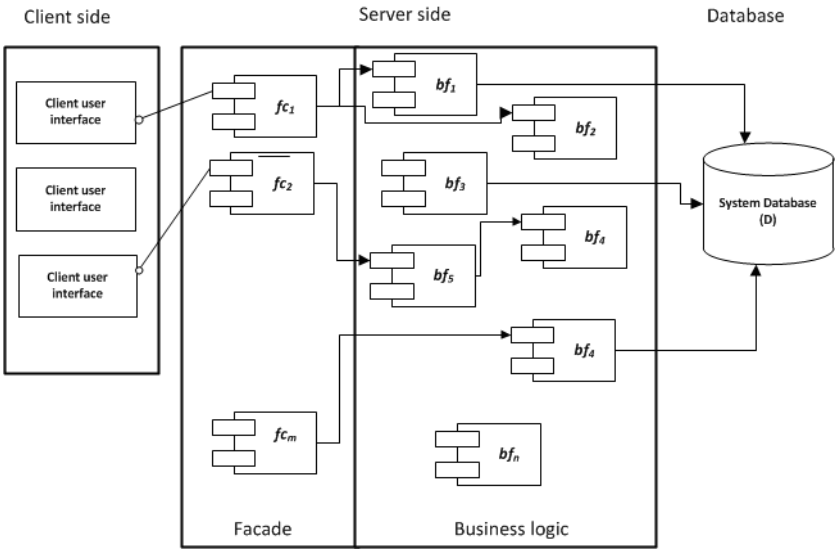


Figure 1. Monolithic application

In formal terms, we assume that a system $\mathtt{S}$ is represented by a triple $(\mathtt{F}, \mathtt{B}, \mathtt{D})$, where $\mathtt{F} = \{\mathtt{fc}_1, \mathtt{fc}_2, \ldots, \mathtt{fc}_{n'}\}$ is a set of facades, $\mathtt{B} = \{\mathtt{bf}_1, \mathtt{bf}_2, \ldots, \mathtt{bf}_{n''}\}$ is a set of business functions, and $\mathtt{D} = \{\mathtt{tb}_1, \mathtt{tb}_2, \ldots, \mathtt{tb}_{n'''}\}$ is a set of database tables. Facades ($\mathtt{fc}_i$) are the entry points of the system that call business functions ($\mathtt{bf}_i$). Business functions are methods that encode business rules and depend on database tables ($\mathtt{tb}_i$). It is also important to define an enterprise organization $\mathtt{O} = \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_w\}$ is divided into business areas $\mathtt{a}_i$, each responsible for a business process.

We describe our technique to identify microservices on a system $\mathtt{S}$ in the following steps:

Step #1: Map the database tables $\texttt{tb}_\texttt{i} \in \texttt{D}$ into subsystems $\texttt{ss}_\texttt{i} \in \texttt{SS}$. Each subsystem represents a business area ($\texttt{a}_\texttt{i}$) of organization $\texttt{O}$. For instance, as presented in Figure 2, subsystem $\texttt{SS}_2$, which represents business area $\texttt{a}_2$, depends on database tables $\texttt{tb}_3$ and $\texttt{tb}_6$. Tables unrelated to business process—e.g., error messages and log tables—are classified on a special subsystem called *Control Subsystem* (SSC).

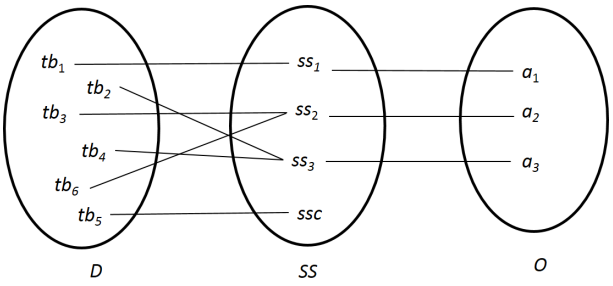


Figure 2. Database decomposition

Step #2: Create a dependency graph (V, E) where vertices represent facades ($\texttt{fc}_\texttt{i} \in \texttt{F}$), business functions ($\texttt{bf}_\texttt{i} \in \texttt{B}$), or database tables ($\texttt{tb}_\texttt{i} \in \texttt{D}$), and edges represent: (i) calls from facades to business functions; (ii) calls between business functions; and (iii) accesses from business functions to database tables. Figure 3 illustrates a graph where facade $\texttt{fc}_2$ calls business function $\texttt{bf}_2$ (case i), business function $\texttt{bf}_2$ calls business function $\texttt{bf}_4$ (case ii), and business function $\texttt{bf}_4$ accesses database tables $\texttt{tb}_2$ and $\texttt{tb}_3$ (case iii).

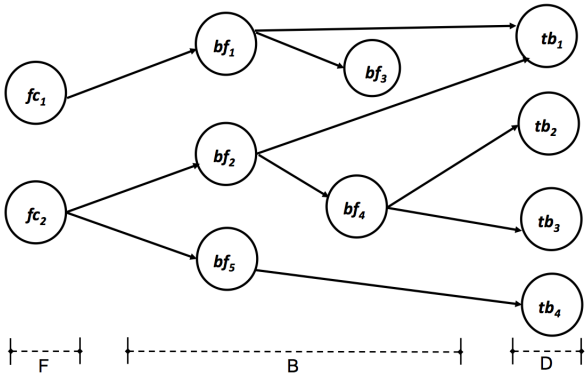


Figure 3. Dependency Graph

Step #3: Identify pairs $(\texttt{fc}_\texttt{i}, \texttt{tb}_\texttt{j})$ where $\texttt{fc}_\texttt{i} \in \texttt{F}$ and $\texttt{tb}_\texttt{j} \in \texttt{D}$, and there is a path from $\texttt{fc}_\texttt{i}$ to $\texttt{tb}_\texttt{j}$ on the dependency graph. For instance, in the graph illustrated in Figure 3, we identify the pairs $(\texttt{fc}_1, \texttt{tb}_1)$, $(\texttt{fc}_2, \texttt{tb}_1)$, $(\texttt{fc}_2, \texttt{tb}_2)$, $(\texttt{fc}_2, \texttt{tb}_3)$, and $(\texttt{fc}_2, \texttt{tb}_4)$.

Step #4: For each subsystem $\texttt{ss}_\texttt{i}$ previously defined in Step #1, select pairs $(\texttt{fc}_\texttt{i}, \texttt{tb}_\texttt{j})$ identified on the prior step where $\texttt{tb}_\texttt{j} \in \texttt{ss}_\texttt{i}$. For instance, in our illustrative example, $\texttt{ss}_3 = \{\texttt{tb}_2, \texttt{tb}_4\}$ then we select the pairs $(\texttt{fc}_2, \texttt{tb}_2)$ and $(\texttt{fc}_2, \texttt{tb}_4)$.

Step #5: Identify candidates to be transformed on microservices. For each distinct pair $(\texttt{fc}_\texttt{i}, \texttt{tb}_\texttt{j})$ obtained on the prior step, we inspect the code of the facade and business func-

tions that are on the path from vertex $\texttt{fc}_\texttt{i}$ to $\texttt{tb}_\texttt{i}$ in the dependency graph. For instance, for pair $(\texttt{fc}_2, \texttt{tb}_2)$, we inspect facade $\texttt{fc}_2$ and business functions $\texttt{bf}_2$ and $\texttt{bf}_4$. The inspection aims to identify which business rules actually depend on database table $\texttt{tb}_\texttt{j}$ and such operations should be described in textual form as rules. Therefore, for each pair $(\texttt{fc}_\texttt{i}, \texttt{tb}_\texttt{j})$, a candidate microservice (M) is defined as follows:

- *Name*: the service name according to pattern [subsystemname].[processname]. For instance, Order.TrackOrderStatus.
- *Purpose*: one sentence that describes the main business purpose of the operation, which is directly associated to the accessed database entity domain. For instance, *track the status of a customer order*.
- *Input/Output*: the data the microservice requires as input and produces as output—when applied—or expected results for the operation. For instance, microservice Order.TrackOrderStatus requires as input the *Order Id* and produces as output a *List of finished steps with conclusion date/time* and a *List of pending steps with expected conclusion date*.
- *Features*: the business rules the microservice implements, which are described as a verb (action), an object (related to the database table), and a complement. For instance, *Identify the order number*, *Get the steps for delivery for type of order*, *Obtain finished steps with date/time*, and *Estimate the date for pending steps*.
- *Data*: the database tables the microservice relies on.

Step #6: Create API gateways to turn the migration to microservices transparent to clients. API gateway consists of an intermediate layer between client side and server side application. It is a new component that handles requests from client side—in the same technology and interface as $\texttt{fc}_\texttt{i}$—and synchronizes calls to the new microservice M and to $\texttt{fc}'_\texttt{i}$—a new version of $\texttt{fc}_\texttt{i}$ without the code that was extracted and implemented on microservice M. An API gateway should be defined for each facade.

There are three cases of synchronization we have to consider in our evaluation: (i) when the input of $\texttt{fc}'_\texttt{i}$ is the output of M or the input of M is the output of $\texttt{fc}'_\texttt{i}$; (ii) when the input of M and $\texttt{fc}'_\texttt{i}$ are the same as API gateway input and the instantiation order is irrelevant; and (iii) when we have to split $\texttt{fc}_\texttt{i}$ into two functions $\texttt{fc}'_\texttt{i}$ and $\texttt{fc}''_\texttt{i}$ and microservice M must be called after $\texttt{fc}'_\texttt{i}$ and before $\texttt{fc}''_\texttt{i}$.

On one hand, if we can synchronize the calls as described on case (i) or (ii), we identify the proposed microservice M as a "*strong candidate*". On the other hand, if we can only synchronize the calls as in case (iii), we identify the proposed microservice as a "*candidate with additional effort*". Particularly in our technique, assuming a microservice of a subsystem $\texttt{ss}_\texttt{x}$, if we identify a business rule in the microservice definition that needs to update data in $\texttt{tb}_\texttt{i} \in \texttt{ss}_\texttt{x}$ and $\texttt{tb}_\texttt{j} \notin \texttt{ss}_\texttt{x}$ in the same transaction scope, we identify the proposed microservice as a "*non candidate*".

When every evaluated pair $(\texttt{fc}_\texttt{i}, \texttt{tb}_\texttt{j})$ of a subsystem is classified as microservice candidate, we recommend to migrate the entire subsystem to the new architecture. In this case, we have to implement the identified microservices, create an independent database with subsystem tables, develop API gateways, and eliminate the subsystem implementation (source code and tables) from system S. Although API gateways must be deployed

in the same server as system S to avoid impacts on client side layer, the microservices can be developed using any technology and deployed wherever is more suitable.

3. Evaluation

We applied our proposed technique on a large system from a Brazilian bank. The system handles transactions performed by clients on multiple banking channels (Internet Banking, Call Center, ATMs, POS, etc.). It has 750 KLOC in the C language and runs on Linux multicore servers. The system relies on a DBMS with 198 tables that performs, on average, 2 million transactions a day.

Step #1: We identified 24 subsystems including subsystem SSC. Table 1 shows a fragment of the result obtained after this initial step. Headers represents subsystems and their content represent the tables they rely on. One problem we identified is that certain tables—highlighted in grey—are associated to more than one subsystem.

Table 1. Mapping of Subsystems and Tables

Business Actions	Service Charges	Checks	Clients	Current Accounts	Saving Accounts	Social Bennefits	Pre-approved credit	Debit and Credit cards	SMS Channel
ACO	AGT	CHS	CLT	*CNT*	*CNT*	BEN	LPA	CMG	CTS
ACB	ISE	TCE		CCT	CPO	DPB		INP	STS
RCA	PTC	ECH		*RCC*	*LAN*	DBC		NPP	CMS
	PTF	HET		*LAN*	*RCC*	IBS		BIN	RCS
	RTE	CCF		CCO	MPO	LBE		CCM	RLS
	RTT			CCE	PPO			PBE	RTS
	TPT			CHE	SPA				
	TTE								
	UTM								
	DUT								

Step #2: We created a dependency graph composed by 1,942 vertices (613 facades, 1,131 business functions, and 198 database tables), 5,178 edges representing function calls and 2,030 edges corresponding to database table accesses. Due to the size of our evaluated system, we proceed our evaluation to the following five subsystems: *Business Actions*, *Service Charges*, *Checks*, *Clients* and *SMS channel*. Table 2 illustrates the characteristics of each subsystem. For subsystem *Business Actions*, we obtained the graph presented on Figure 4.

Table 2. Evaluated Subsystems

Subsystem	Business actions	Service Charges	Checks	SMS channel	Client
Tables (vertices)	3	10	5	6	1
Functions (vertices)	5	62	29	138	>150
Function calls (edges)	3	79	14	133	>150
Database accesses (edges)	6	14	22	140	26
Microservices candidates	1	3	8	4	4

Steps #3–#4: Considering subsystem *Business Actions*, we find the following pairs:

$$(\texttt{AUTCCErspSolAutCceNov}, \texttt{ACO}), \quad (\texttt{AUTCCErspSolAutCceNov}, \texttt{RCA}),$$
$$(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACO}), \quad (\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{RCA}),$$
$$(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACB}).$$

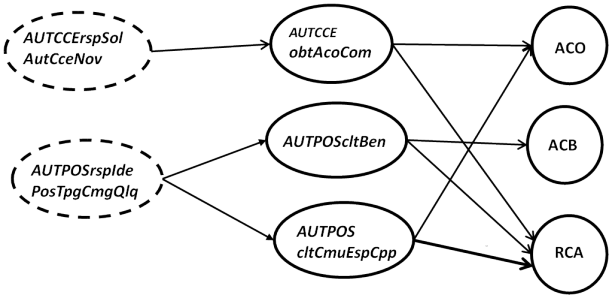


Figure 4. Business Actions Subsystem Graph

Step #5: For pairs $(\texttt{AUTCCErspSolAutCceNov}, \texttt{ACO}), (\texttt{AUTCCErspSolAutCceNov}, \texttt{RCA})$ obtained on prior step, we inspect the code of facade `AUTCCErspSolAutCceNov` and the business functions it calls (`AUTCCEobtAcoCom`), and we identified the microservice described as follows:

- *Name*: `BusinessActions.ListBusinessActionsForAccount`.
- *Purpose*: List business actions related to an account on the relationship channel.
- *Input/Output*: Account number and channel id as input, and a list of business actions as output.
- *Features*: Retrieve business actions assigned to the account; retrieve business actions enabled for the relationship channel; and retrieve the list of business actions assigned to the account and enabled for the channel.
- *Data:* Database tables ACO and RCA.

We also evaluated the other three pairs $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACO})$, $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{RCA})$, $(\texttt{AUTPOSrspIdePosTpgCmgQlq}, \texttt{ACB})$ and the business functions they call `AUTPOScltBen` and `AUTPOScltCmuEspCpp`, and we identified the same microservices as the one described above. In fact, table `ACB` could be merged with `ACO`.

Step #6: For facades `AUTCCErspSolAutCceNov` and `AUTPOSrspIdePosTpgCmgQlq`, we identified an API gateway that suits case (i) described in the proposed technique.

We also evaluated steps #3 to #6 for subsystems: *Service Charges*, *Checks*, *Clients* and *SMS channel*. Although subsystem *Service Charge* has 10 tables and 51 facades, we only identified and defined the following three microservices: `ServiceCharge.CalculateServiceCharge`, `ServiceCharge.IncrementServiceChargeUsage`, and `ServiceCharge.DecrementServiceChargeUsage`. On one hand, we identified 14 API gateways that also suit case (i). On the other hand, we identified other 37 API

that suit case (ii). However, we can avoid the development of the last 37 API gateways since the called microservices have only input data and can be implemented with an asynchronous request. Thus, particularly in this case, we suggest to use a message queue manager (MQM) for communication and substitute the C code that performs an update on database table for a "put" operation on a queue.

For subsystems *Checks* and *SMS channel*, we identified microservices and APIs with the same characteristics of subsystem *Service Charge*, which indicates that both subsystems are good candidates to be migrated to microservices. Nevertheless, for subsystem *Client*—which accesses only one table—we identified one microservice that must be called by more than 50 API gateways that suits case (iii). Therefore, we did not recommend its migration to microservices, since the effort to split and remodularize more than 50 functions, create and maintain more than 50 API gateway are probably greater than the benefits of microservice implementation.

Last but not least, we disregard subsystems that have one or more tables that appear in more than one subsystem list, such as table `CNT` of subsystem *Current Account* (refer to Table 1) because our technique to identify microservices considers that only one subsystem handles operation on each table.

Brief discussion: We classify our study as well-succeeded because we could identify and classify all subsystems, and create and analyze the dependency graph that helped considerably to identify microservices candidates. As our practical result, we recommended to migrate 4 out of the 5 evaluated subsystems to a microservice architecture.

4. Related Work

Sarkar et al. described a modularization approach adopted on a monolithic banking system but they did not use a microservice architectural approach [8]. Richardson evaluated a microservice architecture as a solution for decomposing monolithic applications. His approach considered the decomposition of a monolithic system into subsystems by using use cases or user interface actions [6]. Although this is an interesting approach, in some situations a use case represents a set of operations of different business subsystems which are synchronized by an actor action. Therefore, we do not always have an entirely system described with use cases. By contrast, our technique starts by evaluating and classifying the database tables into business subsystems, which demands access only to the source code and the database model. Namiot et al. presented an overview of microservices architecture but they do not evaluate the concepts on a real-world system [4]. Terra et al. proposed a technique for extracting modules from monolithic software architectures [9]. This technique is based on a series of refactorings and aims to modularize concerns through the isolation of their code fragments. Therefore, they do not target the extraction of modules that can be deployed independently of each other, which is a distinguish feature of microservice-based architectures.

5. Conclusion and Future Work

This paper describes a technique to identify microservices on monolithic systems. We successfully applied the proposed technique on a 750 KLOC real-world monolithic banking system, which demonstrated the feasibility of our technique to identify upper-class

microservices candidates on a monolithic system. The graph obtained for each subsystem helped considerably to evaluate the functions, identify, and describe microservices.

There are subsystems that were not classified as good candidates to migrate to microservices, however. We found scenarios that would require a considerable additional effort to migrate the subsystem to a set of microservices. For instance, (i) subsystems that share same database table, (ii) microservice that represents an operation that is always in the middle of another operation, and (iii) business operations that involve more than one business subsystem on a transaction scope (e.g., money transfer from a check account to a saving account).

More important, the migration to the microservice architecture can be done incrementally. In other words, we can profit from the microservices architecture—e.g., services being developed and deployed independently, and technology independence— without migrating the entire system to microservices. Both kinds of systems architecture—monolithic and microservices—can coexist in a system solution. In fact, one challenge in using microservices is deciding when it makes sense to use it, which is exactly the ultimate goal of the technique proposed in this paper.

As future work, we plan to evaluate other subsystems of the banking system, implement the identified microservices, and measure in the field the real benefits. We also plan to evaluate the proposed technique on Java monolithic systems.

Acknowledgment

Our research has been supported by CAPES, FAPEMIG, and CNPq.

References

[1] Melvin E Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[2] Martin Fowler and James Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014.

[3] M Lynne Markus and Cornelis Tanis. The enterprise systems experience-from adoption to success. *Framing the domains of IT research: Glimpsing the future through the past*, 173:207–173, 2000.

[4] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[5] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[6] Chris Richardson. Microservices: Decomposing applications for deployability and scalability. http://www.infoq.com/articles/microservices-intro, 2014.

[7] Chris Richardson. Pattern: Microservices architecture. http://microservices.io/patterns/microservices.html, 2014.

[8] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35, 2009.

[9] Ricardo Terra, Marco Tulio Valente, and Roberto S. Bigonha. An approach for extracting modules from monolithic software architectures. *IX Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–18, 2012.

Software Evolution Sonification

Pedro O. Raimundo, Sandro S. Andrade, Renato Novais

GSORT Distributed Systems Group
Federal Institute of Education, Science, and Technology of Bahia
Av. Araújo Pinho, 39. Canela. Salvador – Bahia

{pedrooraimundo, sandroandrade, renato}@ifba.edu.br

Abstract. *Program comprehension is one of the most challenging tasks undertaken by software developers. Achieving a firm grasp on the software's structure, behavior and evolution directly from its development artifacts is usually a time-consuming and challenging task. Software visualization tools have effectively been used to assist developers on these tasks, motivated by the use of images as outstanding medium for knowledge dissemination. Under such perspective, software sonification tools emerge as a novel approach to convey temporal and concurrent streams of information due to the their inherently temporal nature. In this work, we describe how software evolution information can be effectively conveyed by audio streams; how music, software architecture concepts and techniques come together to achieve such means; and present a framework for sonification of extracted evolutionary metrics from software repositories.*

1. Introduction

Comprehending computer programs is a notoriously difficult task that involves gathering information from diverse sources (source code, documentation, runtime behavior, version history, just to mention a few) and gets progressively harder as the program's size and complexity grow [Stefik et al. 2011]. Synthesizing that information to tackle the development process effectively and efficiently is an endeavor that requires time, experience and, more often than not, peer support.

Tools are usually employed in order to help the decision making and understanding of the developers. Such tools range from built-in Integrated Development Environment (IDE) helpers and code metric viewers to complex software visualization solutions, focusing on conveying information to the user through the computer screen using tables, charts, drawings or animations due to their higher apprehension, if compared with a naive representation of the same data.

While such approaches are helpful in the comprehension process, aural representations of software structure and behavior have been shown to excel at representing structural [Vickers 1999], relational [Berman 2011] and parallel or rapidly changing behavioral data [Sonnenwald et al. 1990] even for individuals without extensive musical background [Berman 2011, Vickers and Alty 2002, Vickers and Alty 1998].

Software evolution adds another dimension to the problem since it forces the subject to add another layer of understanding to the structural aspects of a software program – the temporal layer. Common analysis requires, for example, information about the relationship between components of a system and lower-level information such as lines of code and fan-in/fan-out values per component. Evolutionary analysis can, for instance,

consume the results of the regular analysis procedures to yield higher-level information such as architectural drift in a given time slice, through the use of the project's meta-data (number of commits, bugs reported, time between releases, etc). Evolutionary analysis can also provide environment-related information which is particularly useful to manage large program evolution and continuing change [Lehman 1980] in computer systems.

In order to convey such information through the means of aural representations it is possible to use any kind of sound. Vickers and Alty proposed that these constructs are more effective if they follow culturally-appropriate styles, conceived analogously to words, which carry meaning to an individual native to or familiar with a specific language. Furthermore, they also found that western musical forms (based on the seven-note diatonic scale) are more readily recognized around the world [Vickers and Alty 2002].

We present a novel approach to transmit information about software evolution by exploiting sound's uniquely temporal nature and aural events such as melody, harmony and rhythm. Different events are used because each event has a distinct impact on the listener and they can be mixed and matched together to convey different streams of information, without being confusing or overwhelming.

The key contributions of this work are as follows. First, we propose a sonification framework in which the evolutionary aspects of a piece of software can be represented as sound streams, in an unobtrusive and noninvasive way. Second, we ran initial validation studies to assert that sonifications carry sufficient meaning to represent the evolutionary aspects of both large and small software projects.

The rest of this work is organized as follows. Section 2 details the pre-existing technologies and techniques utilized to reify the artifacts of this work, and summarizes the developed procedure. Section 3 briefly enumerates some of the implementation details and tools adopted in the framework's development. Section 4 details the validation procedure and presents an analysis of the obtained results. Section 5 summarizes the content and purpose of this paper, epitomizes the main ideas presented herein, and highlight some venues of future studies and experiments in this area.

2. Proposed Approach

This paper proposes a sonification framework that sonifies software evolution by following three main steps: source code retrieval, data extraction, and sound synthesis. The following sections detail the solution's general architecture and the sonification process.

2.1. Proposed Architecture

Figure 1 depicts the structural architectural view of the proposed framework, presenting three specialized software modules and the interactions between them and their shared working set. This approach allows each module to evolve and adapt independently, as long as the format of their shared data and their communication interfaces remains unchanged, and also enables independent reuse of each module. Details on the actual sonification process can be found in Sections 2.5 and 3.

2.2. Source Code Retrieval

Version control systems (VCS) create repositories by storing an initial version of the source code and then successive versions of the files by using delta compression. This

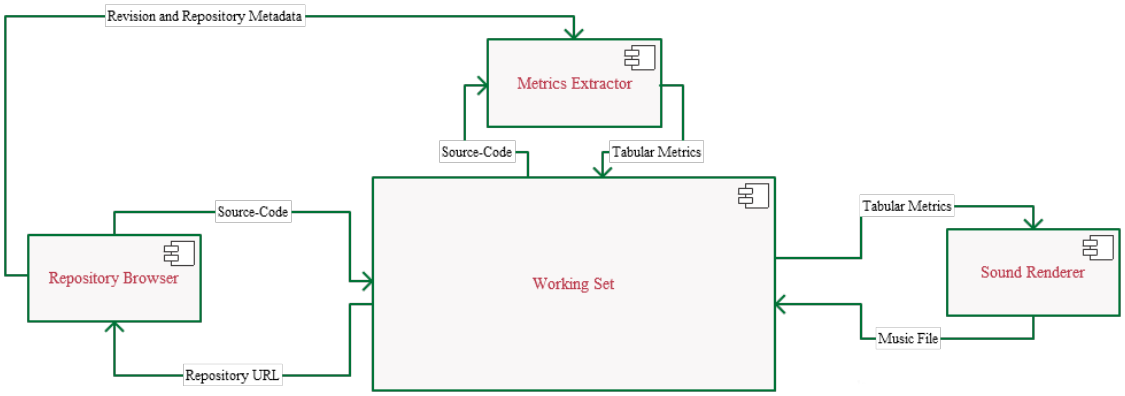


Figure 1. Structural view of the proposed framework.

allows for efficient storage and seamless switching between different snapshots of the versioned project.

Software evolution is an inherently temporal phenomenon. As such, it is fundamental to track the changes in a program's structure and code-metrics across a period of time to achieve a proper representation of it. This involves retrieving snapshots of the software's source code at different revisions. VCS greatly simplify this task since repositories created by such tools can be systematically transversed, processed and compared.

In order to extract meaningful data from software repositories and generate interesting sonifications, the metrics extractor module uses filters. While the framework includes some filters to demonstrate its capabilities, the ability to easily write and combine new filters is the highlight that allows the user to transverse source code repositories in order to achieve virtually any goal.

2.3. Data Extraction

The broad field of investigations that generally deal with extracting data from software repositories in order to uncover trends, relationships and extract pertinent information is called Mining Software Repositories (MSR). Software repositories refer, in this context, to the entirety of development artifacts that are produced during the process of software development and evolution (usually excluding by-products of the building process).

The content of these aggregated data sources exists throughout the entirety of the project's life cycle and carries a wealth of information that includes but is not limited to: the versions that the system has gone through, meta-data about the revisions of the software (as seen in Subsection 2.2), the rationale for project's architectural choices and discussions between the project's members. In this work, the focus is not on answering questions through MSR but using it to display the software evolutionary aspects focusing on the changes to properties rather than internal measuring.

Our approach does not, though, contemplate the specialized software evolution metrics developed by Lehman and Ramil in [Ramil and Lehman 2000]. The rationale is that those metrics focus heavily on cost-estimation and applied aspects of project management, whereas this work focuses on program comprehension.

2.4. Sound Synthesis

Martin Russ [Russ 2009] defines Sound Synthesis as:

> (...) the process of producing sound. It can reuse existing sounds by processing them, or it can generate sound electronically or mechanically. It may use mathematics, physics or even biology; and it brings together art and science in a mix of musical skill and technical expertise (...)

This is a broad, but sufficient, definition for the purposes of this work, in which the ultimate goal is to electronically generate sounds that are both meaningful and musical.

The artistic vein of sound and music allows sonifications to rouse specific emotions on the listener, the displays of technical expertise and emotion by musicians also greatly affect the impression left on the listener. While raw sound synthesis lacks these qualities, Vickers and Alty [Vickers and Alty 2013] go to great lengths to assert that the aesthetics and structural motifs of music itself determine the effects of sonifications on readers; along with cultural and psychological aspects and specific preferences of each subject.

In this work, sound is produced electronically through a tool for musical notation that utilizes its own Domain-Specific Language. This approach ensures that the aural metaphors developed can be systematically associated with the numbers that represent software metrics, without the element of *musical performance*.

2.5. Summarized Procedure

Working with the building blocks detailed above, the sonification process we propose herein can be summarized as the following automatic steps:

1. Download a source-code repository through a version control system;
2. Extract the desired metrics from the repository's meta-data or several versions of the source-code;
3. Process the extracted metrics along with a pre-defined sonification template.

Some implementation details of this process and the adopted technologies are detailed in the next section.

3. Implementation

The proposed framework was developed using the Java programming language, chosen due to its good balance of productivity, debuggability and the pre-existence of several libraries and components necessary for this implementation.

The initial implementation of the framework adopted Git as the Version Control System, the FreeMarker engine [Revusky et al. 2015] to process the sonification templates and the GNU Lilypond music notation tool [Nienhuys and Nieuwenhuizen 2015] to generate the aural output with the desired characteristics.

In order do run the exploratory study, presented in Section 4, the first implementation of the framework is a simple Java application that, given the URL of a Git repository, opens such a repository locally (downloading it over the Internet if not already present),

Figure 2. Graphical User Interface of the runner class.

extracts the desired metrics for a predefined period and generates the corresponding sonification of the extracted data in the *.ly* file format. Then, Lilypond processes this file and generates both MIDI and PDF files with the sonification results.

This implementation includes extractors for two project metrics: commits per month and committers per month. In the context of open-source software, these two metrics provide insight on the overall health and the community around a software project.

The sonification template included maps the commits per month metric to a specific pitch, and the number of committers per month to a number of notes. Subsection 4.2 elaborates further on how this is represented on the finished sonification.

4. Validation (Exploratory Study)

A exploratory study was conducted to validate the proposed framework. The experimental procedure, their goals and our findings are detailed in the following subsections

4.1. Goals

The exploratory study was undertaken in order to assert whether or not the sonifications rendered from evolutionary data gathered from source-code repositories are meaningful and easily understood. Additionally, this exploratory study helps determine whether or not the basic implementation of the framework has enough assets to render useful sonifications.

4.2. Procedure

For this exploratory study, two *open-source* office suites were selected as subjects: the KOffice suite – whose developed ceased in 2013 and the LibreOffice suite – whose development is still well underway and is largely utilized by the open-source community.

We selected the period from *01/01/2010* to *31/04/2013* and generated a sonification of the interpolated evolutionary metrics. The seemingly arbitrary finish date for the sonification period corresponds to the month *before* the last non-automatic commit of the KOffice project, because KOffice's activity ceased midway through the month, whereas LibreOffice continued all throughout.

The monthly number of commits in each project was mapped to the pitch representing that specific month, while the number of committers for a given month corresponded to the number of notes it lasts for, as specified by the default template file. The extracted data was interpolated to make sure the minimum and maximum number of commits correspond to the pitches of C2 (two octaves below the middle C) and C8 (four octaves above the middle C), while the minimum and maximum number of committers correspond to, respectively, 2 and 8 notes; such interpolation strategy was utilized in the extractor classes to represent both projects in a similar perspective, even through their

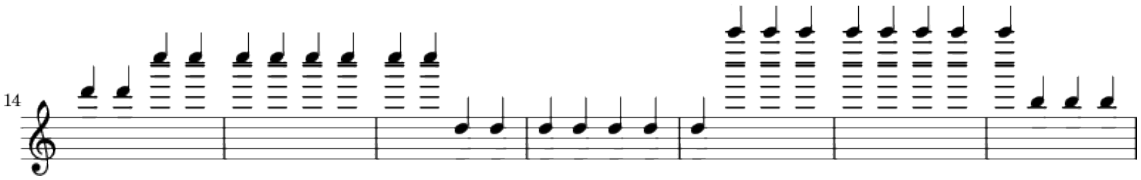


Figure 3. Snippet of the musical score for LibreOffice's sonification.

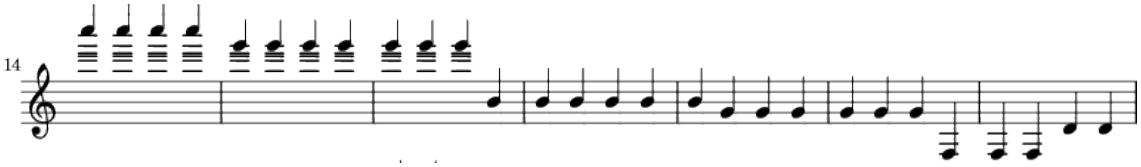


Figure 4. Snippet of the musical score for KOffice's sonification.

numbers were in slightly different orders of magnitude. Snippets of the musical scores for the sonifications can be seen in Figures 3 and 4.

The finished sonifications were qualitatively analyzed in order to determine if the desired mappings were correctly reproduced in the sonification, if both projects were represented in a similar fashion, and if it is possible to get an insight on the project's health and activity through sonification alone. It should be noted that the musical scores are provided here as a printed alternative to the sound output, which is the focus of the work.

4.3. Results

A musical analysis of the sonification results is in order to further explain the impact of the elected aural metaphors in the process of program comprehension. In Figure 5, an annotated version of the previously shown snippet of LibreOffice's evolution is provided to support this analysis process.

First, let's take a look on the differences in pitch and their expected effects in the user of the auralizations. The blue and yellow highlights in Figure 5 correspond to the months with the least and the most commits, respectively; it is expected that this mapping feels the most natural to any individuals familiar with western forms of music, where higher tones are usually employed to rouse stimulant effects and lower tones inspire serenity and quietude.

The second aural event utilized in this mapping, number of notes, corresponds to the amount of people involved in a month's commits, and determines how many notes with the same pitch will be played in succession. This metaphor is closely related to

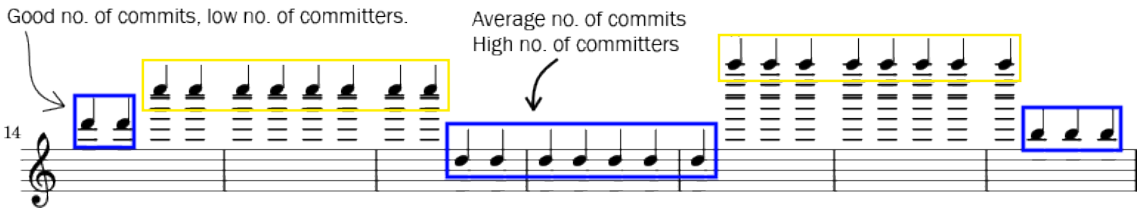


Figure 5. Anotated snippet of the musical score for LibreOffice's sonification.

sound's inherently temporal nature and, once trained, users should quickly internalize that more notes equals more people.

It should be noted that no additional effort is necessary to convey or understand temporal aspects in sonification. Once a user is trained to understand a specific set of metrics and aural events, the passing of the time is immediately perceived upon exposure to the sonifications.

One apparent shortcoming of these mappings is that two consecutive months with similar amounts of commits and different amounts of committers will show up as a single longer string of notes with similar pitch, this is by design because these larger repeated strings are *promptly* detected by users and may highlight a strong burst of activity or inactivity, a very important moment in the projects history or both. Months could be artificially separated by pauses or other musical elements, this was avoided until further empirical results confirm or deny the importance of this event.

While further experiments are necessary to confirm the effects of these mappings, previous studies have shown that musical auralizations with western characteristics are promptly apprehended by listeners of diverse backgrounds with or without previous musical training [Berman 2011, Vickers and Alty 2002].

Despite the project's numbers being in different orders of magnitude (LibreOffice has always had a lot more activity than KOffice), the sonifications were able to represent both projects in a comparable scale. If a software project evolves quickly and ascends to a larger scale, the latest sonification generated should be the one taken into consideration, since it will accurately represent this event by attributing lower pitches and less notes to the previous months, in contrast with the higher values of most recent ones. No additional training should be required to make use of the new sonifications as long as the mappings and metrics itself do not change.

Through analysis of the sonifications, some conclusions were drawn. First, it is possible to coherently map data to aural events. Second, through sonification it is possible to analyze large and small software projects under a similar perspective, preserving the evolutionary trends of each project and investing roughly the same amount of effort for each project.

From an evolutionary standpoint, the sonifications evidenced what could be an important pattern. In KOffice's sonification there were mostly extreme frequencies, meaning that there were many moments of heavy development and long periods of very modest development. In contrast, LibreOffice's sonification had mostly moderate frequencies, with the pitches varying in a roughly wavelike pattern, even in its most extreme moments, meaning that development sprints were cyclical and well-defined. More projects should be analyzed in order to confirm or deny the validity of these patterns. The *wav* files for the evolution sonification of LibreOffice and KOffice projects are available at `http://wiki.ifba.edu.br/sonification`.

5. Conclusion

In this work, we presented some of the pre-existing technologies and techniques that give grounds to the implementation of a sonification framework for software evolution, discussed several implementation details of the proposed tool and performed initial val-

idation of the framework and its results, we ultimately conclude that sound is a medium worth investigating for the transmission of evolutionary aspects of software.

Future efforts planned in this line of work include a systematic literature review of all the studies that were revealed in this work, to foster future research efforts in the field, and further experimental works to assert the effectiveness of the proposed sonification strategy, minimize the risks to the validation presented here, and try to uncover further evolutionary patterns that are better exposed by aural metaphors.

References

Berman, L. (2011). *Program Comprehension Through Sonification*. PhD thesis, Durham University.

Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

Nienhuys, H.-W. and Nieuwenhuizen, J. (1996–2015). GNU LilyPond. `http://lilypond.org/`.

Ramil, J. F. and Lehman, M. M. (2000). Metrics of software evolution as effort predictors - A case study. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*, pages 163–172. IEEE Computer Society.

Revusky, J., Szegedi, A., and Dékány, D. (2002–2015). FreeMarker. `http://freemarker.org/`.

Russ, M. (2009). Chapter 1 - Background. In Russ, M., editor, *Sound Synthesis and Sampling (Third Edition)*, Music Technology, pages 3 – 86. Focal Press, Oxford, third edition edition.

Sonnenwald, D., Gopinath, B., Haberman, G., Keese, W., and Myers, J. (1990). *InfoSound: An audio aid to program comprehension*, volume 2, pages 541–546. Publ by Western Periodicals Co.

Stefik, A., Hundhausen, C. D., and Patterson, R. (2011). An empirical investigation into the design of auditory cues to enhance computer program comprehension. *Int. J. Hum.-Comput. Stud.*, 69(12):820–838.

Vickers, P. (1999). *CAITLIN : implementation of a musical program auralisation system to study the effects on debugging tasks as performed by novice Pascal programmers*. PhD thesis, Loughborough University.

Vickers, P. and Alty, J. (1998). Towards some organising principles for musical program auralisations. In *Proceedings of the Fifth International Conference on Auditory Display*.

Vickers, P. and Alty, J. L. (2002). Musical program auralisation: a structured approach to motif design. *Interacting with Computers*, 14(5):457–485.

Vickers, P. and Alty, J. L. (2013). The well-tempered compiler? the aesthetics of program auralization. *CoRR*, abs/1311.5434.

Como o formato de arquivos XML evolui? Um estudo sobre sua relação com código-fonte

David S. França[1], **Eduardo M. Guerra**[1], **Maurício F. Aniche**[2]

[1]Laboratório de Matemática e Computação Aplicada
Instituto Nacional de Pesquisas Espaciais (INPE) – São José dos Campos, SP – Brasil

[2]Departamento de Ciência da Computação
Universidade de São Paulo (USP), São Paulo - Brasil

{davidsfranca,guerraem,mauricioaniche}@gmail.com

Abstract. *There are different ways to integrate applications. A popular approach is the exchange of messages in XML format where both applications share the XML schema. This model, known as "contract", may change over time as structural modifications such as additions of new information. Changes like that propagate to the client applications, as they have to fit to the new structure and contract. The aim is to better understand how are the dynamics of the contract changes in a software project by means of analyzing messages contracts changes and their impact on the source code of open source and industry applications.*

Resumo. *Existem diferentes formas de integrar aplicações. Uma delas é a troca de mensagens em formato XML, onde ambas aplicações compartilham o modelo estrutural do formato daquele XML. Este modelo, chamado de "contrato", pode sofrer alterações ao longo do tempo, como modificações estruturais ou agregação de novas informações, fazendo com que as aplicações que o utilizam tenham que se adequar a nova estrutura sempre que o contrato é alterado. O objetivo é entender melhor como é a dinâmica de alterações de contratos em arquivos XML por meio de uma análise nas alterações de contratos de mensagens e seu impacto sobre o código-fonte de aplicações de código aberto e da indústria.*

1. Introdução

Com o crescente aumento da quantidade de informações disponíveis e a necessidade de troca desses dados entre aplicações, diversas soluções para integração de sistemas são utilizadas, adotando as mais diversas tecnologias encontradas no mercado. Segundo [Kalin 2013], é raro que um sistema de software funcione inteiramente de modo isolado. O típico sistema de software deve interagir com os outros sistemas, que podem estar em diferentes máquinas, serem escritos em diferentes linguagens, ou mesmo utilizarem diferentes plataformas.

Com esta finalidade de integração, umas das soluções é a utilização de arquivos *Extensible Markup Language*(XML) para troca de mensagens. A utilização desse de arquivo traz a vantagem de ser moldável às necessidades das aplicações, onde "elementos são arranjados em uma estrutura hierárquica, similarmente a uma árvore, que reflete tanto a estrutura lógica dos dados quanto a estrutura de armazenamento"[Bates 2003]. Muitas

vezes a integração entre as aplicações necessita de garantia na validade das informações, para isso, arquivos XML são validados segundo uma estrutura definida em um modelo estrutural de formato de dados que é compartilhado pelas aplicações chamado também de contrato. [Meyer 1997] diz que contrato "é relação entre uma classe e seus clientes como um acordo formal, expressando direitos e obrigações de cada parte". O padrão mais utilizado para a criação contratos de validação de arquivos XML é o *XML Schema Definition* (XSD). Nele são descritos, por meio de sua linguagem própria, qual a exata estrutura que um XML deve ter para ser considerado válido como, por exemplo, nomes das entidades, sua multiplicidade e opcionalidade. [Valentine 2002] dizem que, em outras palavras, o *schema* define as restrições para um vocabulário XML.

No entanto, as aplicações que compartilham um modelo na troca de mensagens podem sofrer alterações ao longo do desenvolvimento do projeto, fazendo com que as outras aplicações, que compartilham o mesmo modelo, não consigam mais estruturar suas mensagens no formato acordado. Para tanto, a cada mudança no modelo de mensagens todas as aplicações que o utilizam devem alterar suas implementações para criar mensagens no formato. [França and Guerra 2013] dizem que a evolução de aplicações quando um contrato é alterado dá-se de forma lenta e difícil, alterando-se apenas parte das aplicações por vez ou em velocidades diferentes. Assim, este trabalho tem por objetivo fazer um estudo acerca das mudanças nos contratos compartilhados por aplicações que desejam se integrar e qual é a relação de uma mudança desse contrato no código-fonte das aplicações. Para isso, quatro questões de pesquisa foram investigadas a fim de tentar entender a utilização de modelos de mensagens e sua relação com o código-fonte das aplicações:

(Q1) Qual é a frequência de modificações em arquivos XSD?

(Q2) Quais os tipos mais comuns de modificações em arquivos XSD?

(Q3) Qual a relação entre alterações em arquivos XSD e alterações no código-fonte?

(Q4) As alterações do XSD costumam ficar concentradas em um desenvolvedor?

Este artigo apresenta na seção 2 a metodologia de pesquisa, na seção 3 o estudo sobre projetos industriais, na seção 4 o estudo sobre os projeto de código aberto. Na seção 5 faz-se uma análise entre os dois estudos. Na seção 6 apresentam-se os trabalhos relacionados. Na seção 7 apresentam-se as conclusões e nas seções 7 e 8 apresentam-se as limitações encontradas na pesquisa e as conclusões do estudo..

2. Metodologia de pesquisa

O objetivo do estudo é extrair métricas a partir dos *commits* de projetos e analisar as alterações feitas em arquivos XSD, bem como sua relação com mudanças no código fonte. A análise dos dados tenta responder uma série de questões pertinentes às mudanças de arquivos XSD como quantidade de alterações, frequência de alterações, tipos de mudanças efetuadas, número de desenvolvedores que alteraram arquivos XSD e a relação dessas alterações com arquivos de código-fonte.

Para isso foram feitos dois estudos em dois grupos de projetos: industriais e de código fonte aberto. Inicialmente, foram adquiridos os dados referentes aos *commits* de

cada projeto através dos dados armazenados nos controladores de versões dos projetos. Utilizaram-se duas formas de obtenção de dados: manualmente, onde cada *commit* de alteração de arquivos de contratos foi avaliado; e outra de forma automática, utilizando uma ferramenta de mineração de dados. Para cada *commit* de alteração de arquivos de contrato, foi obtida a data de alteração, quem o alterou, o número de arquivos de código-fonte no mesmo *commit*, além das médias de alterações de arquivos XSD por projeto e a média dos arquivos de código-fonte alterados no mesmo *commit* de arquivos XSD.

Para tipos de mudança, em cada alteração no contrato verificou-se, comparando o arquivo antes e depois, o que mudou nele, podendo assim, distinguir qual foi a alteração sofrida e dividi-las quanto ao tipo de alteração. Para o caso da frequência de modificações, foi extraído de cada *commit* sua data, gerando uma série temporal de mudanças para cada arquivo. Quanto a sua relação com o código fonte, quando um *commit* possui um arquivo XSD, contou-se o número de arquivos de código-fonte também neste mesmo *commit*. Vale ressaltar que no caso da análise da relação com o código-fonte nos projetos industriais, esta contagem foi feita de forma manual, verificando no momento da recuperação de cada *commit* se os arquivos de codígo-fonte tem relação coma mudança no XSD. Esta análise foi feita de modo empírico, por desenvolvedores das aplicações. Por fim, comparou-se os dois conjuntos de métricas na esperança de se obter evidências a respeito do comportamento de tais arquivos independentemente do projeto avaliado. Essa comparação abrangeu a média de arquivos de código-fonte ligados a mudança de contrato, a frequência de modificações e a distribuição dos integrantes da equipe do projeto.

3. Estudo sobre projetos industriais

Fez-se um levantamento em três projetos que utilizam um mesmo modelo na troca de mensagens. Todos eles são gerenciados pelo controlador de versões *Subversion*. Um dos projetos é o fornecedor dos serviços, que provê dados às outras aplicações. Quanto aos projetos analisados, tratam-se de 3 aplicações da área de defesa que formam um simulador de eventos discretos com aproximadamente 5 anos de desenvolvimento. Na tabela 1 pode-se verificar alguns dados referentes a elas.

Tabela 1. Dados dos projetos industriais analisados

Nome	linhas de código	classes	XSDs	entidades nos XSDs
Fornecedor 1	27199	235	1	114
Consumidor 1	174085	1243	1	114
Consumidor 2	116648	1240	1	114

3.1. Realização do estudo

O primeiro passo é extrair todos os *commits* referentes às mudanças nos arquivos XSD. Para se obter o histórico de *commits* de cada arquivo do contrato (um contrato pode estar dividido em vários arquivos) utilizou-se a ferramenta TortoiseSVN. A recuperação dos dados foi feita de forma manual já que são apenas 3 projetos e 1 contrato compartilhado entre eles. A comparação entre versões do contrato também foi feita de forma manual, isto é, para cada mudança no contrato, obtem-se a diferença antes e depois da alteração e

comparando-o para saber de que tipo de mudança se trata. Nesse passo 95 *commits* foram selecionados para análise.

Desses dados foram extraídas as frequências das modificações, divididas em trimestres, já que os projetos possuem um longo período de desenvolvimento, além da distribuição das mudanças dos contrato entre os desenvolvedores dos projetos. Além disso, uma alteração no contrato da mensagem influencia tanto o fornecedor, que deve alterar seu código-fonte para gerar a mensagem no novo formato, como os consumidores, que devem alterar seus códigos-fonte para consumirem as mensagens no novo formato. Assim foram também extraídos dados em relação ao número arquivos de código-fonte alterados quando um contrato sofre modificações.

4. Estudo sobre projetos de código aberto

A segunda etapa consiste em obter e analisar dados de um repositório de projetos de código aberto com um maior volume de dados. É feito um levantamento em um repositório da USP (Universidade de São Paulo) gerenciado pelo controlador de versões Git, possuindo 67 projetos, dentre eles 22 utilizam diferentes contratos. Para este estudo é utilizada a ferramenta MetricMiner [Sokol et al. 2013], uma aplicação *web* que facilita o trabalho de pesquisadores envolvidos em mineração de dados em repositórios de software. A ferramenta ajuda pesquisadores em todas as etapas durante um estudo de mineração de repositório de código, como a clonagem do repositório, extração de dados, criando conjuntos de dados específicos e executar testes estatísticos. Utilizando as funcionalidades desta ferramenta, foi desenvolvida uma consulta e acoplada a ferramenta de mineração, que retorna apenas os dados avaliados como relevantes na análise.

4.1. Realização do estudo

Foram extraídas métricas referentes ao número de arquivos XSD presentes nos projetos, a frequência em que foram alterados e a distribuição dessas alterações ao longo do tempo. Dos 67 projetos avaliados, 22 contêm arquivos XSD e destes, 11 os alteraram. Foram extraídos tambem dados referentes a média de alterações por mudança em um XSD, além do número de arquivos de código-fonte alterados a cada mudança em um XSD.

5. Análise dos resultados

(Q1)Qual é a frequência de modificações em arquivos XSD? Os estudos indicam que quando um projeto possui arquivos XSD e este é alterado, não ocorre apenas uma vez, mas várias vezes e distribuídos ao longo do período de desenvolvimento do projeto. Isto indica a necessidade de atualização de todas as aplicações que precisam seguir o contrato alterado. No primeiro estudo, dos 19 trimestres analisados, apenas 5 não tiveram alterações no contrato, isto mostra que as outras duas aplicações que compartilham este arquivo, também foram ou deveriam ser atualizadas constantemente para continuar seguindo o contrato, dependendo da necessidade de utilização da nova imformação disponivel pelo contrato e poderem trocar informações entre si, como mostra a Figura 1. No segundo estudo, viu-se que em alguns projetos as modificações chegam a acontecer durante todo o desenvolvimento, como mostra a Figura 2. Alterá-los e replicar essas mudanças em outras aplicações que o utilizam pode ser muito custoso ou até mesmo inviável.

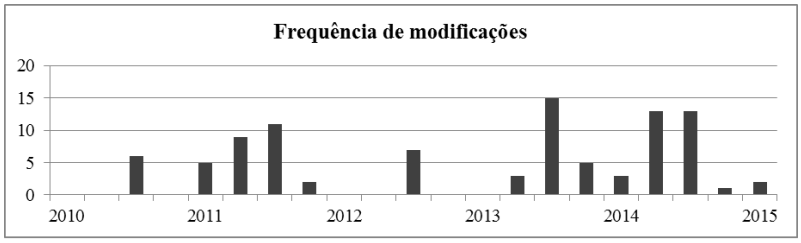


Figura 1. Frequência das modificações nos contratos XSD

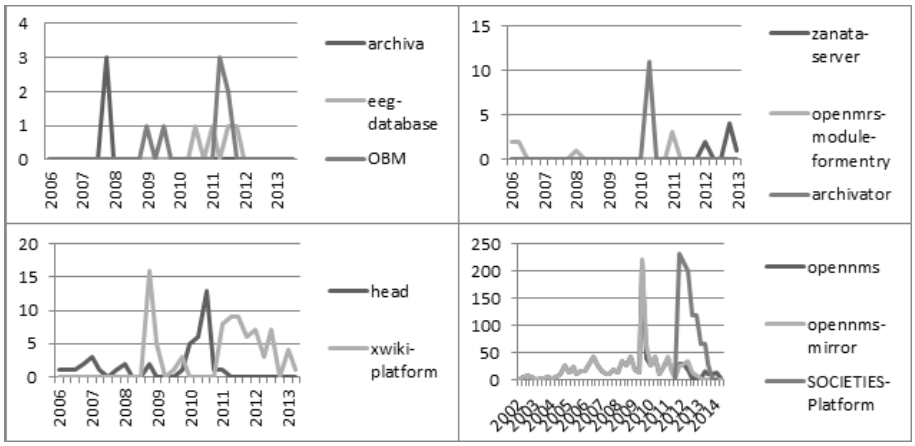


Figura 2. Alteração de arquivos XSD por projeto ao longo do desenvolvimento

(Q2)Quais os tipos mais comuns de modificações em arquivos XSD? Observa-se no primeiro estudo 9 tipos de modificações nos arquivos XSD, onde a adição de um novo elemento e alteração de um elemento são mais comuns como é mostrado na Tabela 2, indicando respectivamente, a inserção de novos dados a uma mensagem e alterações de estrutura ou semântica. Logo, estes tipos de modificações podem quebrar o contrato de mensagens entre as aplicações.

(Q3)Qual a relação entre alterações em arquivos XSD e alterações no código-fonte? Verificou-se que contratos têm certa relação com seu código-fonte. Para cada modificação no contrato é necessário um esforço para adequar a aplicação para ler ou gerar mensagens no novo formato. Pode ser muito custoso quando várias aplicações estão envolvidas ou a equipe é pequena. Vê-se no segundo estudo, que a média de modificações de arquivos de código-fonte quando um contrato é alterado pode ser muito alta, indicando esse tipo de relação, mostrado na Tabela 3. Como segunda evidência deste relação, no primeiro estudo, mostrado na Figura 3, para cada *commit* de alteração no contrato foi verificado classe a classe do código-fonte no mesmo *commit* se esta estava ligada ou não a mudança do contrato. Essa verificação é feita de forma empírica, baseado na experiência dos desenvolvedores do projeto, para se dizer se a alteração da classe está ou não ligada a mudança no contrato. Como uma mudança em um contrato pode não sofrer *commit* juntamente com as alterações no código-fonte, foram considerados os 5 *commits* adjacentes à alteração do contrato, para se tentar garantir que as alterações de código-fonte estavam sendo corretamente contadas. Constatou-se através dessa análise manual que, em média, 43,77% dos arquivos de código-fonte no mesmo *commit* de alteração do contrato estão relacionados a mudanças nos contratos.

Tabela 2. Tipos de Modificações em arquivos XSD

Tipo de Modificação	Quantidade
Adição de um novo elemento	51
Modificação de um elemento	18
Alteração de multiplicidade	15
Alteração de enum	12
Modificação de um atributo	10
Remoção de um elemento	5
Adição de enum	3
Adição de um novo atributo	2
Remoção de um atributo	2

Tabela 3. Estatísticas de modificações em projetos código aberto

Projeto	NAX	MAX	MMAX	MAJ	MMAJ	NC
archiva	1	2	2,00	15	7,50	1
archivator	1	7	7,00	9	1,29	4
eeg-database	1	2	2,00	61	30,50	2
head	5	29	5,80	1943	67,00	12
OBM	1	1	1,00	25	25,00	1
openmrs-module-formentry	2	3	1,50	18	6,00	2
opennms	212	2065	9,74	25743	12,47	37
opennms-mirror	154	2005	13,02	42121	21,01	36
SOCIETIES-Platform	104	1751	16,84	47836	27,32	54
xwiki-platform	5	76	15,20	7700	101,32	14
zanata-server	1	5	5,00	6337	1267,40	4

Legendas: NAX – Nº de Arquivos XSD, MAX – Modificações em Arquivos XSD, MMAX – Média de Modificações em Arquivos XSD, MAJ – Modificações em Arquivos JAVA, MMAJ - Média de Modificações em Arquivos JAVA, NC – Nº de Commiters.

(Q4)As alterações do XSD costumam ficar concentradas em um desenvolvedor? Como mostrado na Figura 4, notou-se uma diferença bastante relevante. Projetos industriais tendem a possuir a mesma equipe de desenvolvimento por um período maior que os de código aberto. Isso foi avaliado na obtenção dos dados, onde os projetos industriais analisados tiveram 11 desenvolvedores alterando o contrato durante os 5 anos de desenvolvimento dos projetos. Já nos projetos de código aberto, em média, tiveram 15,18 desenvolvedores alterando os contratos dos projetos em um tempo menor de desenvolvimento.

6. Trabalhos relacionados

Diferentemente do presente trabalho que trata de formatos de arquivos XML, [Su et al. 2001] relacionam as mudanças dos *XML schemas* de banco de dados XML com *queries* criadas e classifica as alterações tanto no *schema* quanto nas *queries*. No trabalho é dito que um *schema* pode ter nova versão até a cada 15 dias em um projeto e as

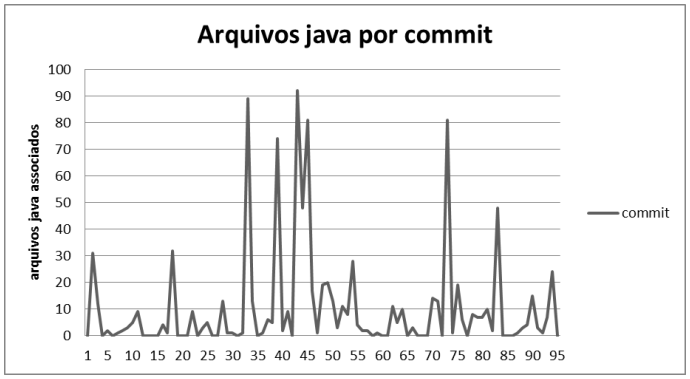


Figura 3. Alteração de arquivos .java por *commit* de alteração no contrato

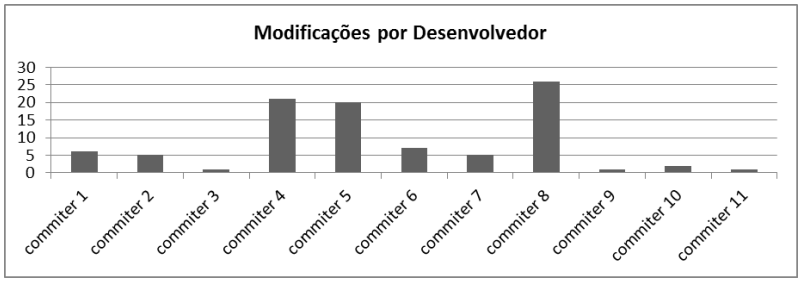


Figura 4. Número de alterações por desenvolvedor

mudanças no XML e nas *queries* são divididas em dois grupos: mudanças basicas e complexas. Por fim, concluiu propondo um conjunto inicial de diretrizes para a preservação de consultas em diferentes versões do esquema.

[Mesiti et al. 2006] fazem uma investigação das mudanças ocorridas em *XML schemas* classificando-as em tipos. Foi também apresentado um conjunto de mudanças primitivas que é a base para qualquer outro tipo de mundança. Esses tipos foram utilizados para propor uma solução de revalidação de XMLs contra *XML schemas* por partes, utilizando mudanças atômicas, livrando assim de ter que se validar tod O presente trabalho também faz uma investigação das mudanças ocorridas nos XML *Schemas*, porém também se preocupa investigar o impacto dessas mudanças sobre o código-fonte.

[Nečaskỳ et al. 2012] versam sobre a evolução de *XML schemas* quando há alteração de requisito. As modificações são mapeadas de acordo com seus espalhamentos horizontais e verticias sobre o sistema. Assim como [Mesiti et al. 2006], ele apresenta um conjunto de mudanças primitivas e o trabalho propõe um *framework* para esse evolução, abrangendo mundanças do tipo *platform-independent model* e *platform-specific model*. O trabalho se preocupa em rastrear as mudanças ocorridas nos modelos quando um requisito é alterado, diferentemente do presente estudo que continua esta linha, partido da alteração do XML *schema* e rastreado seu impacto no código-fonte.

7. Limitações do estudo

XSD de configuração: Uma das limitações do estudo foi considerar as definições de arquivos XML a partir de arquivos XSD de de forma equivalente em toda aplicação, independente do propósito do documento, como, por exemplo, para armazenamento em

disco ou para contratos de serviços *web*. **Poucos projetos**: outro ponto que se deve considerar é a pequena quantidade de projetos de cunho industrial, pois a aquisição de código fonte de projetos deste cunho muitas vezes não é trivial, já que envolvem produtos de empresas que necessitam manter certa privacidade. **Tipo de modificações em código aberto**: outra limitação foi a não investigação dos tipos de alterações dos contratos nos projetos de código aberto, pois demandaria um tempo muito grande neste refinamento da pesquisa. Uma possivel solução é avaliar as mudança pelos comentários dos *commits*.

8. Conclusões

Concluiu-se que são frequentes as modificações dos contratos em ambos os tipos de projetos, e as modificações mais encontradas alteram significantemente a estrutura da mensagem. Essas modificações não são pontuais, sendo diluídas ao longo do desenvolvimento dos projetos. O estudo apresentou indícios que essas modificações geram um forte impacto no código-fonte, fazendo com que uma mudança em um contrato implique na alteração de implementação dos projetos. Concluiu-se ainda que as alterações não estão concentradas em um único desenvolvedor, mas distribuídas por vários integrantes do time.

Quanto a relação de mudanças de XSD com alterações do código-fonte, ficou evidente, no primeiro estudo, que existe a relação de alteração do código-fonte quando um contrato é alterado. Isto complementado pelo segundo estudo que indicou que mudanças em contratos são frequentes ao longo do desenvolvimento.

Referências

Bates, C. (2003). *XML in theory and practice*. J. Wiley, Chichester, England Hoboken, NJ.

França, D. S. and Guerra, E. M. (2013). Modelo arquitetural para evolução no contrato de serviços no contexto de aplicações de comando e controle. *XV Simpósio Internacional de Guerra Eletrônica, SIGE*, 15:23.

Kalin, M. (2013). *Java Web services : up and running*. O'Reilly, Beijing.

Mesiti, M., Celle, R., Sorrenti, M. A., and Guerrini, G. (2006). X-evolution: A system for xml schema evolution and document adaptation. In *Advances in Database Technology-EDBT 2006*, pages 1143–1146. Springer.

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall PTR, Upper Saddle River, N.J.

Nečaskỳ, M., Klímek, J., Malỳ, J., and Mlỳnková, I. (2012). Evolution and change management of xml-based systems. *Journal of Systems and Software*, 85(3):683–707.

Sokol, F. Z., Finavaro Aniche, M., Gerosa, M., et al. (2013). Metricminer: Supporting researchers in mining software repositories. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 142–146. IEEE.

Su, H., Kramer, D., Chen, L., Claypool, K., Rundensteiner, E., et al. (2001). Xem: Managing the evolution of xml documents. In *Research Issues in Data Engineering, 2001. Proceedings. Eleventh International Workshop on*, pages 103–110. IEEE.

Valentine, C. (2002). *XML schemas*. SYBEX, Alameda, Calif.